Réplication et cohérence de données (Data replication and consistency)

Gérald Oster <gerald.oster@loria.fr>

(support de Claudia-Lavinia Ignat)

Course overview

- Introduction to replication
- Consistency models (*)
- Consistency protocols (*)
- Pessimistic replication vs. optimistic replication
- Optimistic replication approaches

(*) Andrew S. Tanenbaum, Maarten Van Steen, "Distributed Systems: Principles and Paradigms“, 2002

Agenda

- Pessimistic replication vs. optimistic replication
- Clocks, logical clocks, state vectors
- Optimistic replication approaches
  - CVS, Subversion
  - Thomas write rule

Pessimistic vs. optimistic replication (1)

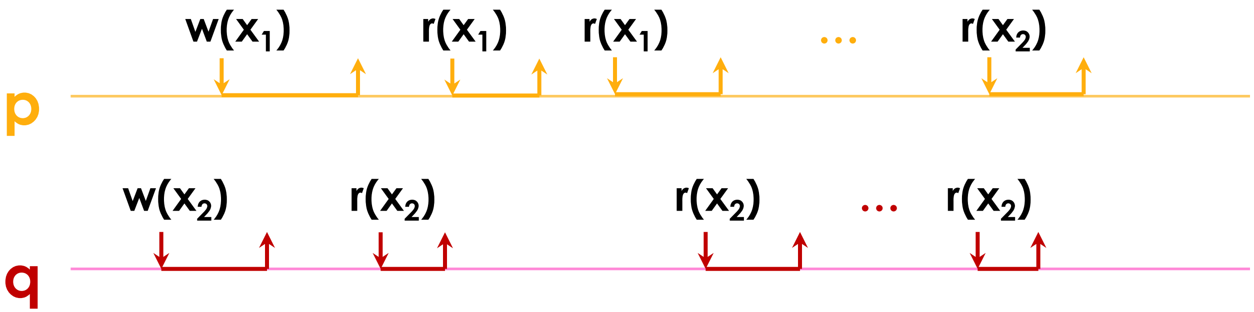
- Pessimistic replication
  - Give the illusion of one replica (no divergence)
  - Block access to a replica unless it is up-to-date
  - Example: primary-copy algorithms
    - Elect a primary replica
    - After an update primary writes the change to secondary replicas
    - If primary crashes elect a new replica
  - Bad performance and availability

Pessimistic vs. optimistic replication (2)

- Optimistic replication
  - Allows replicas to diverge
    - Commit modifications immediately and propagate later
    - Observers can see different values on different sites
  - Eventual consistency
  - Mandatory for offline access
  - Better scaling

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.



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

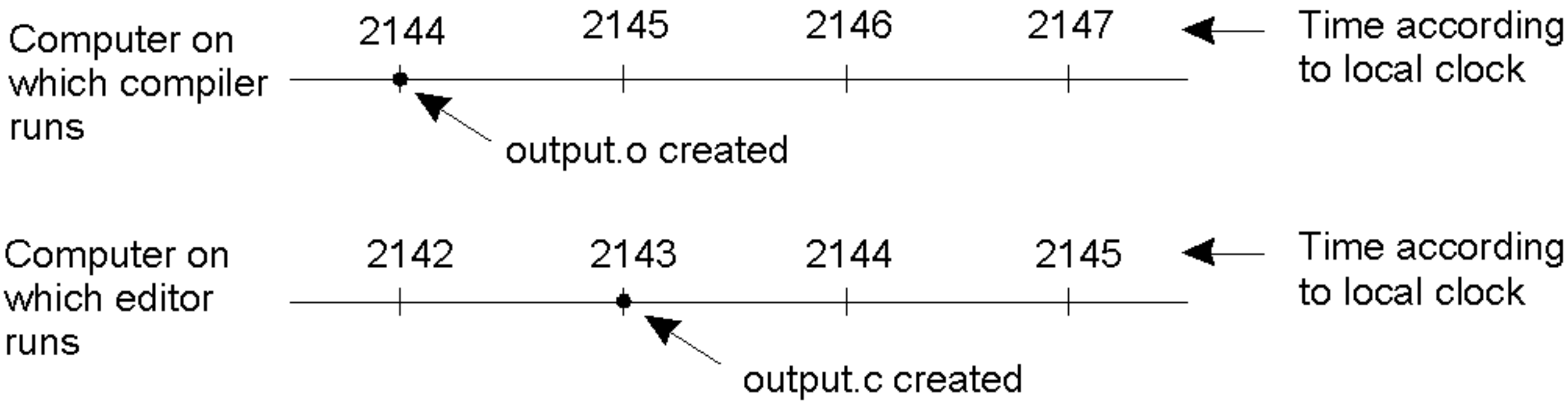
Pessimistic vs. optimistic replication (3)

- Basic principles of (operation-based) optimistic replication
  - N sites replicate an object
  - An object is modified by applying an operation
  - Local operations applied immediately
  - Operations broadcast to the other sites
  - Remote operations integrated and executed
  - System is correct if when it is idle all replicas are identical

Clock Synchronisation

- Time is unambiguous in a centralised system
- There is no global agreement on time in a distributed system
- Example
  - Program consisting of 100 files
  - Use of *make* to recompile only changed source files
  - If input.c has time 2151 and input.o has time 2150, then recompilation needed

Clock Synchronization



- make does not call the compiler

Logical clock

- Sufficient that all machines agree on the same time (not necessarily real time)
- Lamport 1978 – rather than agreeing on what time it is, sufficient to agree on the order in which events occur
- Previous example: if input.c is older or newer than input.o

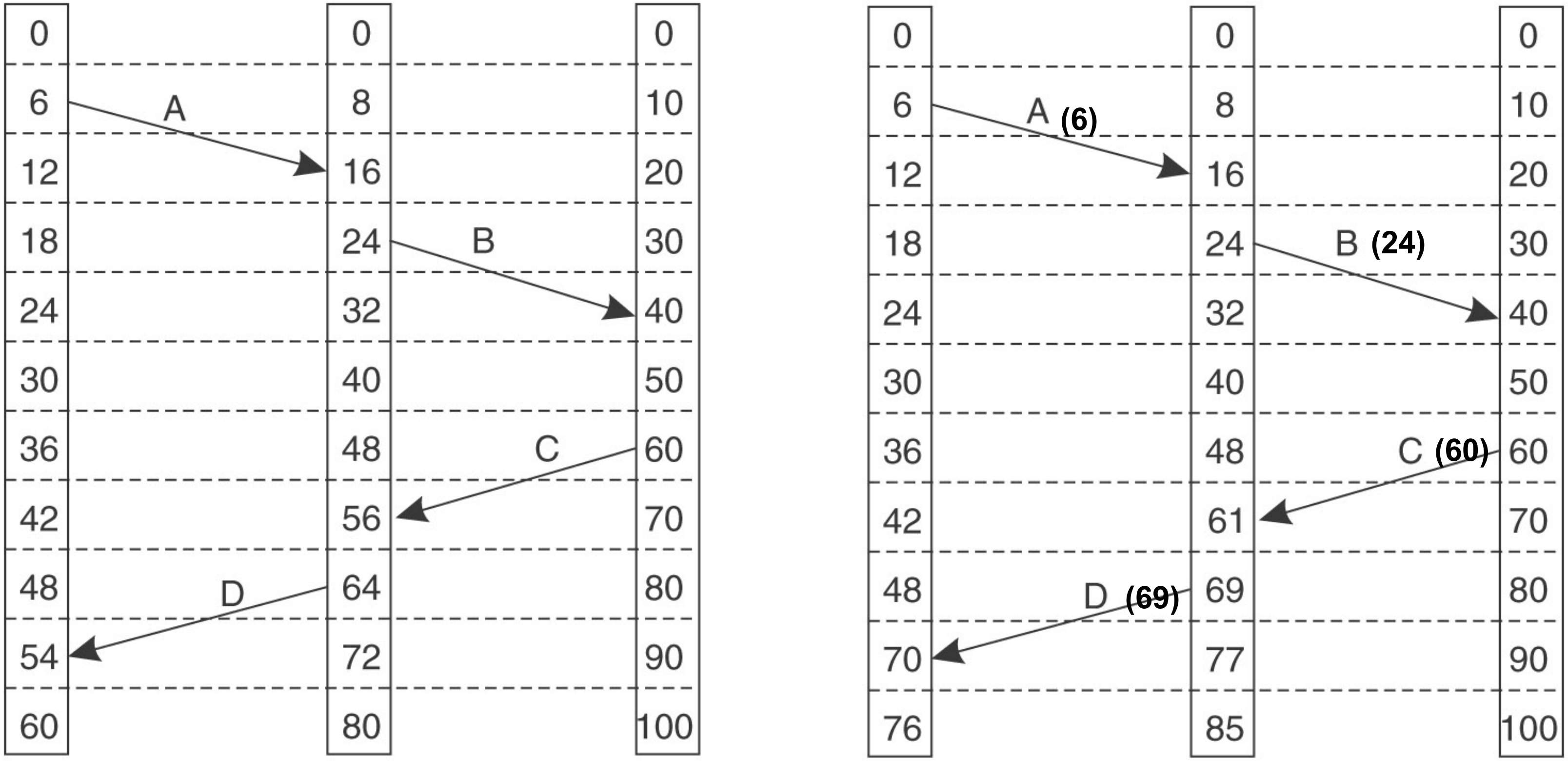
Lamport timestamps

- Happens-before relation
- *a*→*b* (*a* happens before *b*)
- Two situations:
  - If *a* and *b* are events in the same process and *a* occurs before *b*, then *a*→*b*
  - If *a* is the event of a message being sent by one process and b is the event of the message being received by another process, then *a*→*b*. A mesage cannot be received before or at the same time it is sent
- If *a*→*b* and *b*→*c* then *a*→*c*
- If neither *a*→*b* nor *b*→*a* then *a is concurrent with b*

Lamport timestamps

- For every event *a* assign *C(a)* on which all processes agree
- If $a \rightarrow b$ then $C(a) < C(b)$
- Clock time must always increase
- Lamport solution
  - Each message carries the sending time
  - If receiver clock < time of the arrived message, then receiver forwards its clock to 1 + sending time

Lamport timestamps



(a)

(b)

Lamport timestamps

- If *a* happens before *b* in the same process then $C(a)<C(b)$
- If *a* and *b* represent the sending and receiving of a message, $C(a)<C(b)$
- For all distinctive events *a* and *b*, $C(a)\neq C(b)$
  - Attach the number of the process to the lower order of the time
  - If *a* generated by process 1 at time 40 and *b* generated by process 2 at time 40, then $C(a)=40.1$ and $C(a)=40.2$

Vector timestamps

- Lamport timestamps limits
  - if $C(a)<C(b)$ does not imply that $a \rightarrow b$
  - $a \mid\mid b$ does not imply $C(a)=C(b)$
- Example: posting articles and reactions to posted articles
- Lamport timestamps do not capture causality
- Vector timestamps capture causality
  - If VT(a)<VT(b), then a causally precedes b
  - Each process P_i maintains V_i
    - $V_i[i]$ = the no. of events that occured so far at P_i
    - If $V_i[j]=k$ then P_i knows that k events occurred at P_j

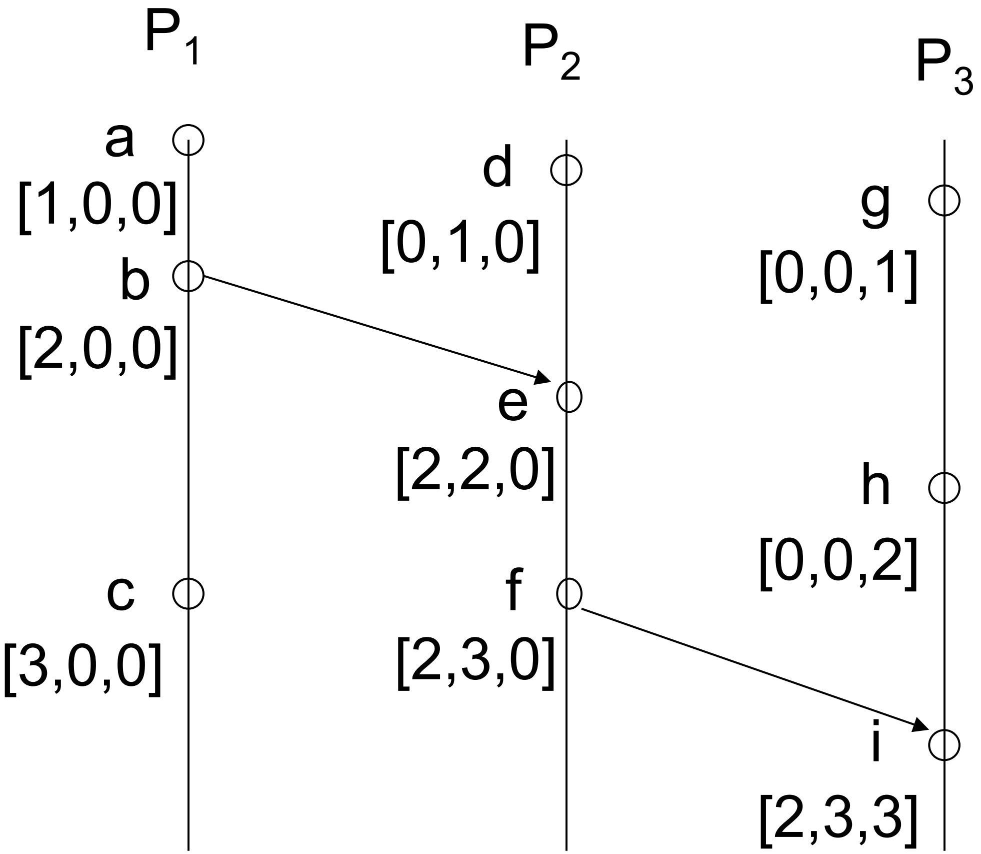
Vector timestamps

- Comparison of two vectors
  - V=W iff ∀ i V[i]=W[i]
  - V<W iff forall i V[i]≤W[i] and ∃i V[i]<W[i]
  - [1,2,0] < [3,2,1]
  - [0,1,1] ≮ [1,0,1]

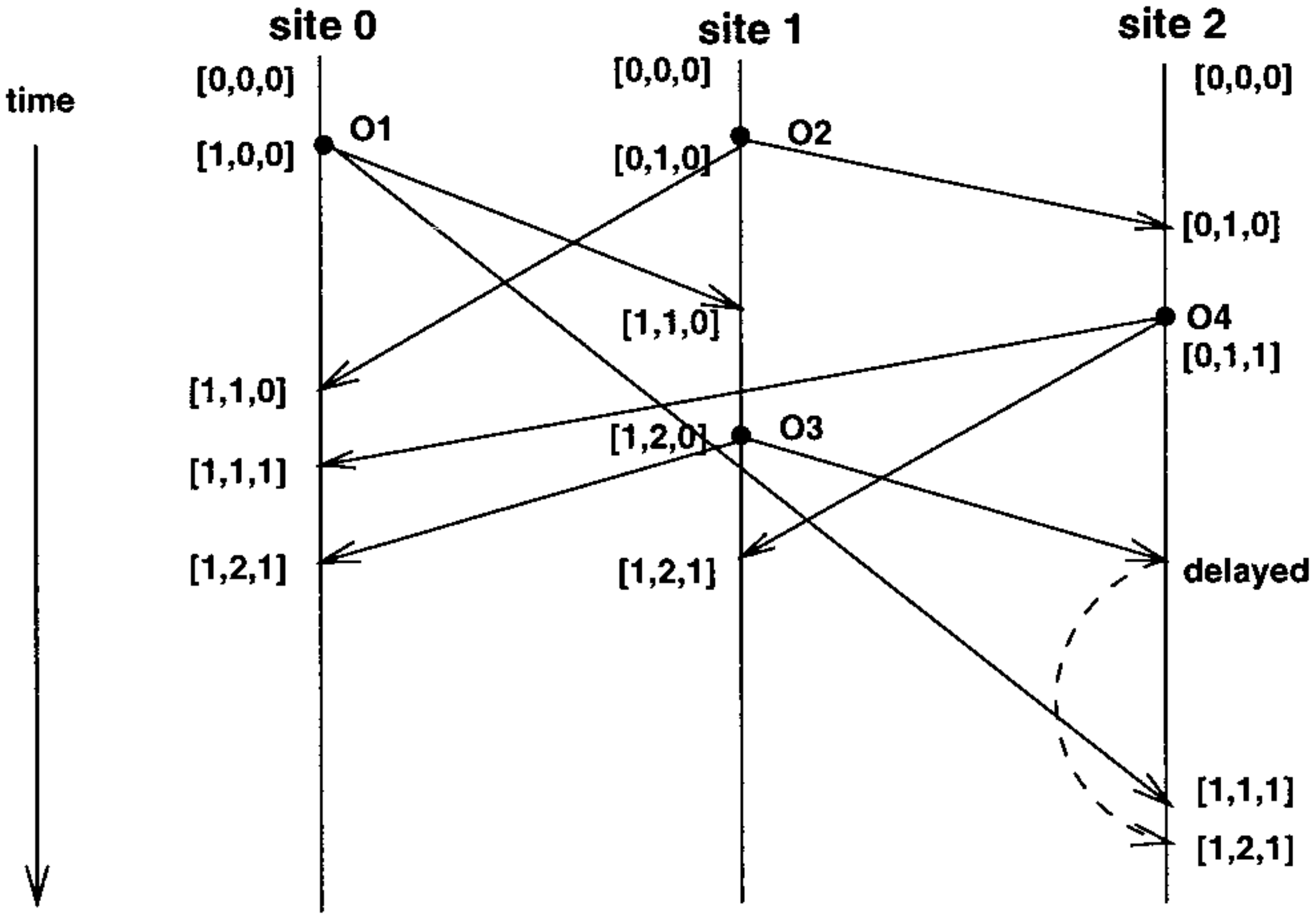
Vector timestamps – computation rules

- Process Pi
  - Initialisation: ∀k Vi[k]=0
  - Local event: Vi[i]= Vi[i]+1
  - Sending message m : Vi[i]= Vi[i]+1, then send (m,Vi)
  - Receiving message (m,Vj):
    - ∀ k Vi[k]=max(Vi[k], Vj[k])
    - Vi[i]=Vi[i]+1

Vector timestamps – example



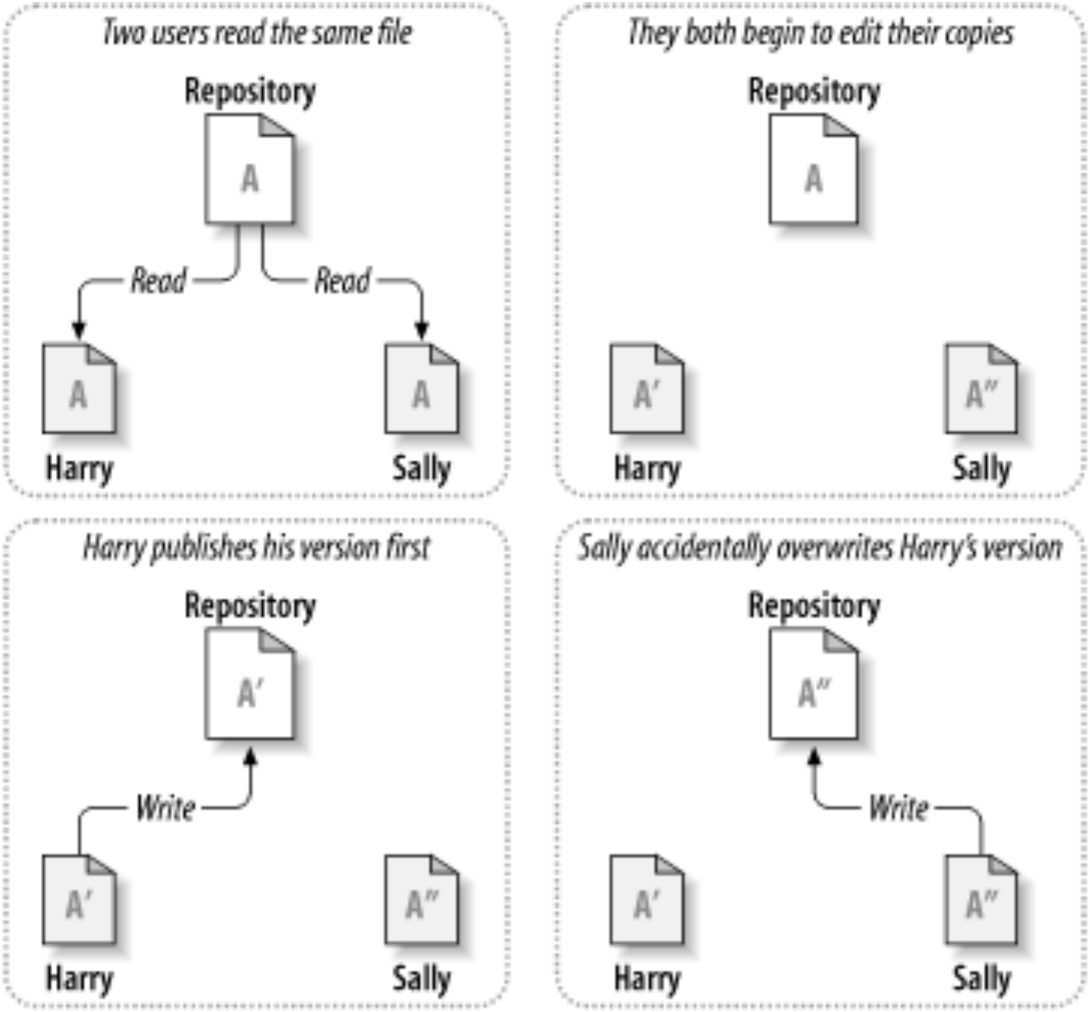
State vector



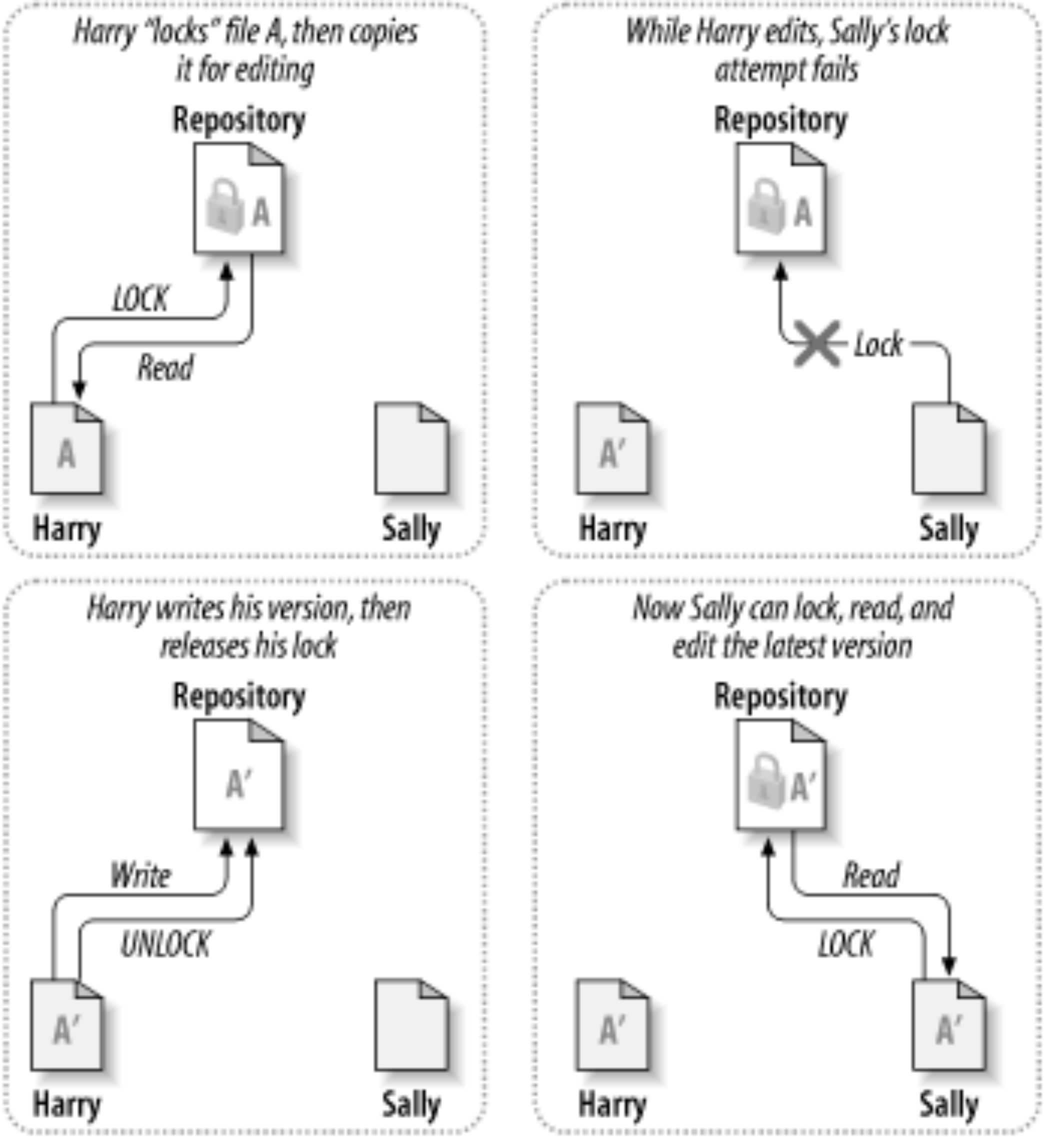
State vector– computation rules

- Process Pi
  - Initialisation: ∀k Vi[k]=0
  - After local execution of an event e: Vi[i]= Vi[i]+1
  - Then, e is timestamped with Vi
  - (e,Vi) will be sent
  - Receiving event (e,Vj):
    - ∀ k Vi[k]=max(Vi[k], Vj[k])

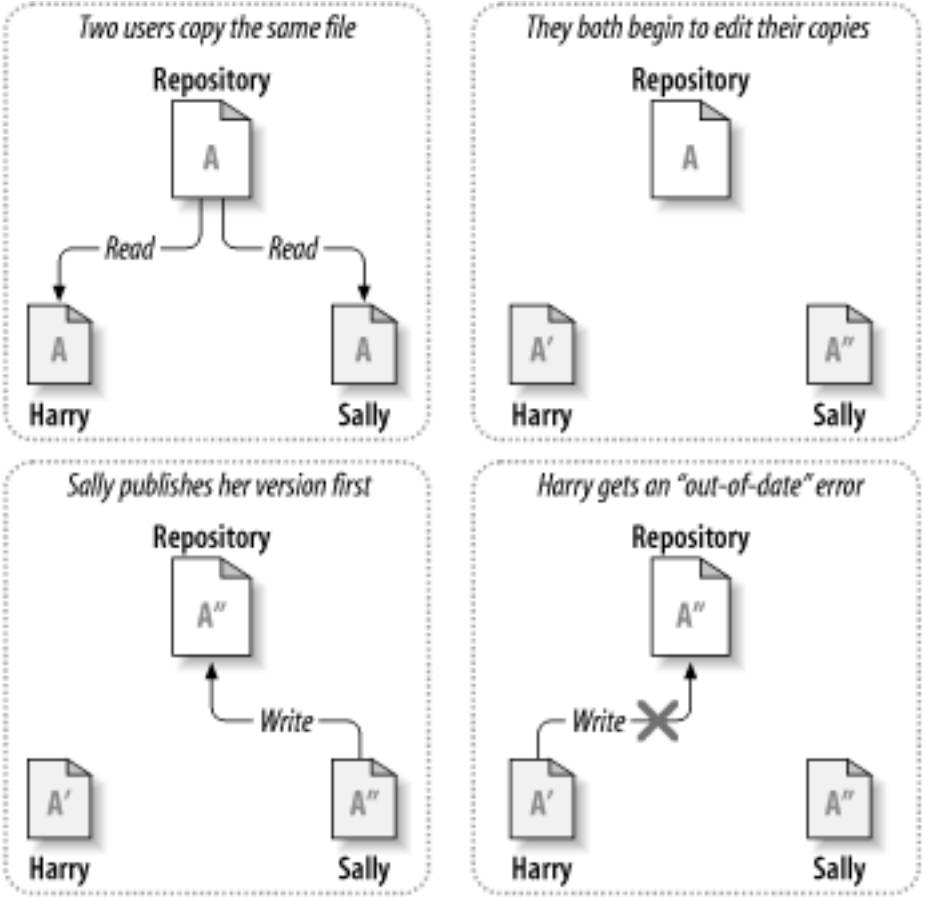
Example: CVS, Subversion



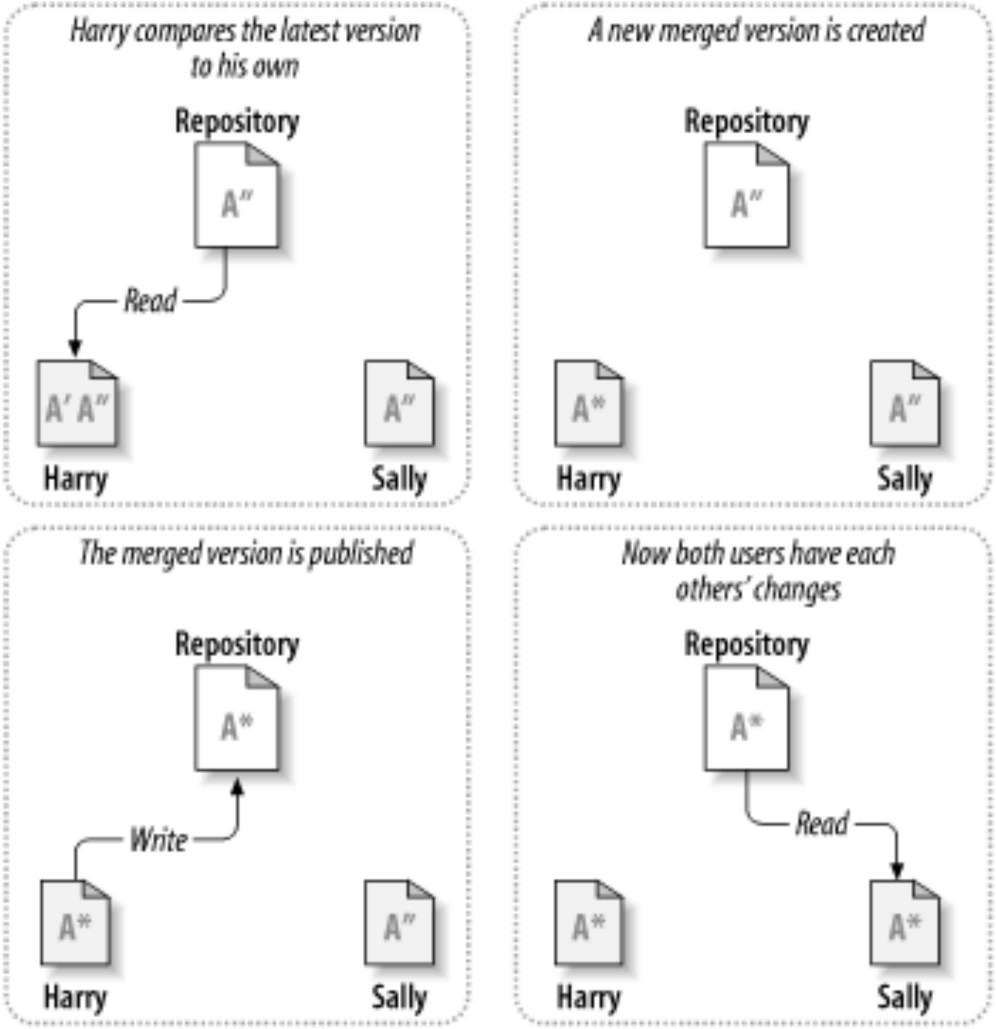
Lock-modify-unlock solution



Copy-modify-merge solution



Copy-modify-merge solution

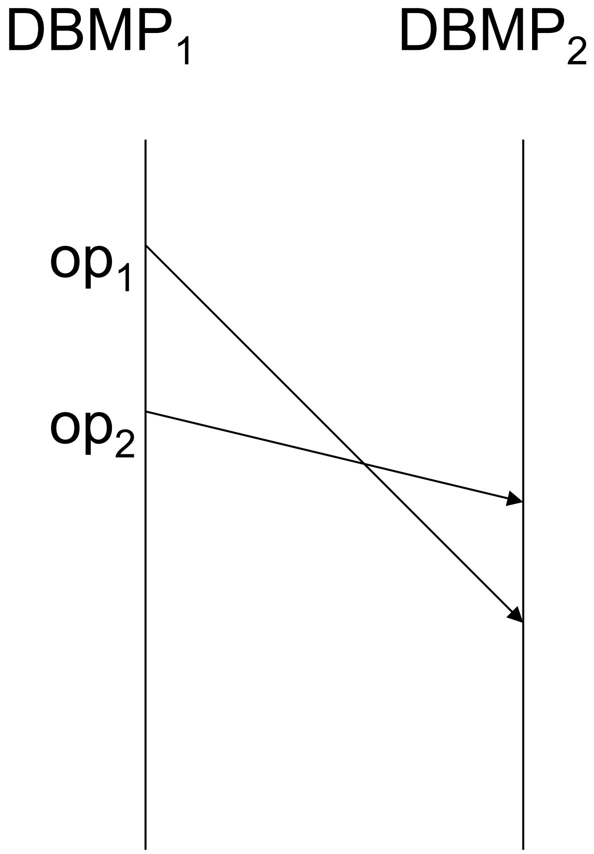


Duplicated databases (Thomas Write Rule 1975) (*)

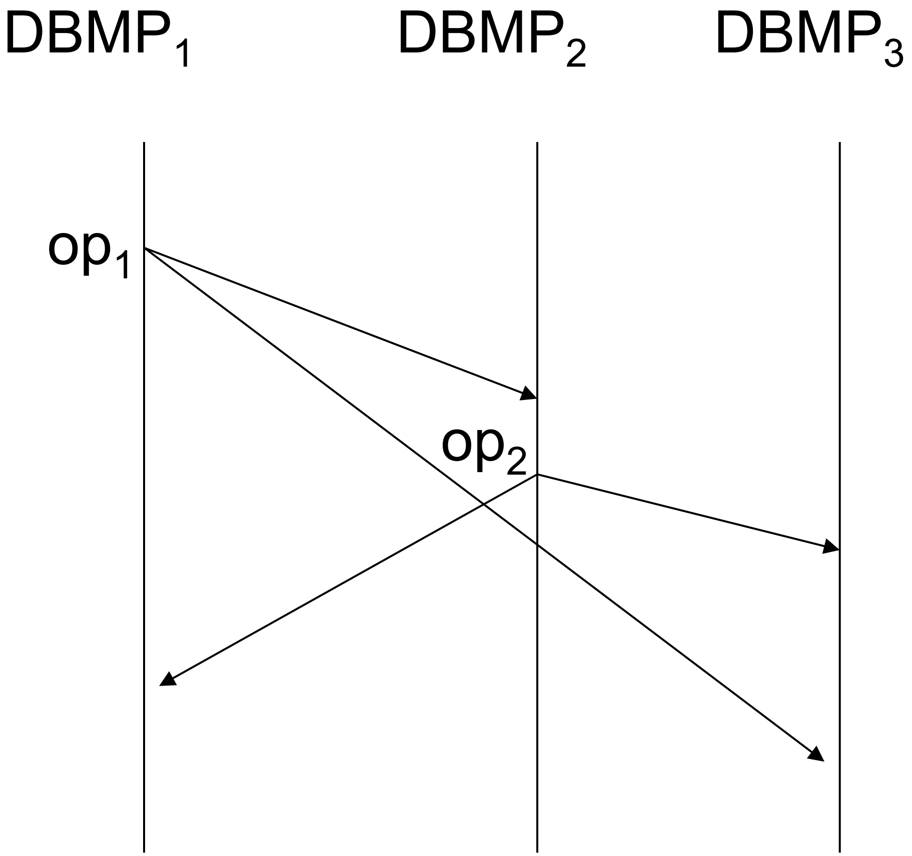
- Model
  - A set of independent DBMPs
  - Each DBMP has its own copy of the database
  - DBMPs communicate via messages
  - Communications are subject to failures
  - Messages between two sites are delivered in the same order they were sent (FIFO)
  - No use of global timestamps
- The system is correct if it eventually converges

(*) P. Johnson and R. Thomas. RFC677 : The maintenance of duplicate databases, 1975.

Duplicated databases (Thomas Write Rule 1975)



Not possible

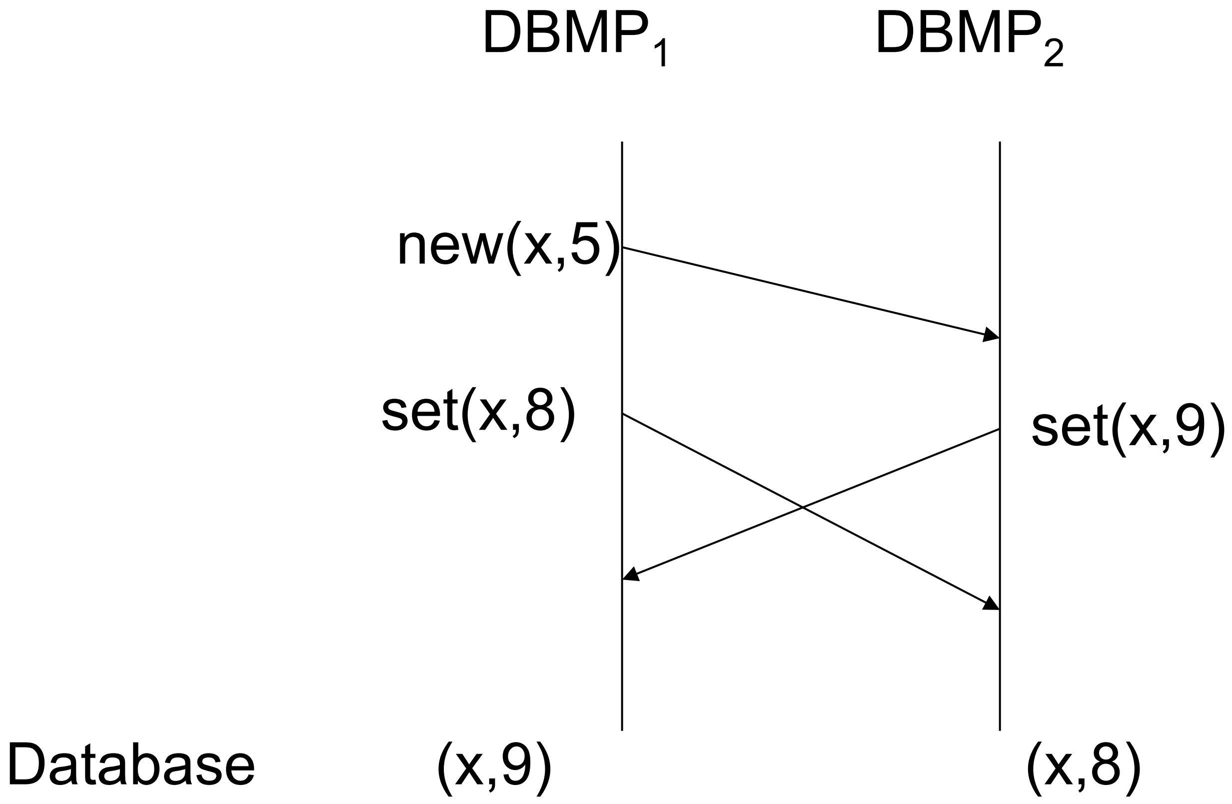


Possible

Duplicated databases (Thomas Write Rule 1975)

- The database = collection of (selector,value) pairs
- Operations:
  - Selection:
    - get(selector) returns the current associated value
  - Assignment:
    - set(selector, new_value) replaces associated value with new_value
  - Creation:
    - new(selector, initial_value) adds (selector, initial_value) entry
  - Deletion:
    - delete(selector, value) deletes existing (selector, value) pair

Duplicated databases (Thomas Write Rule 1975)



DBMP$_1$ DBMP$_2$

new(x,5)

set(x,8)

set(x,9)

Database (x,9) (x,8)

- How to guarantee that copies are consistent?

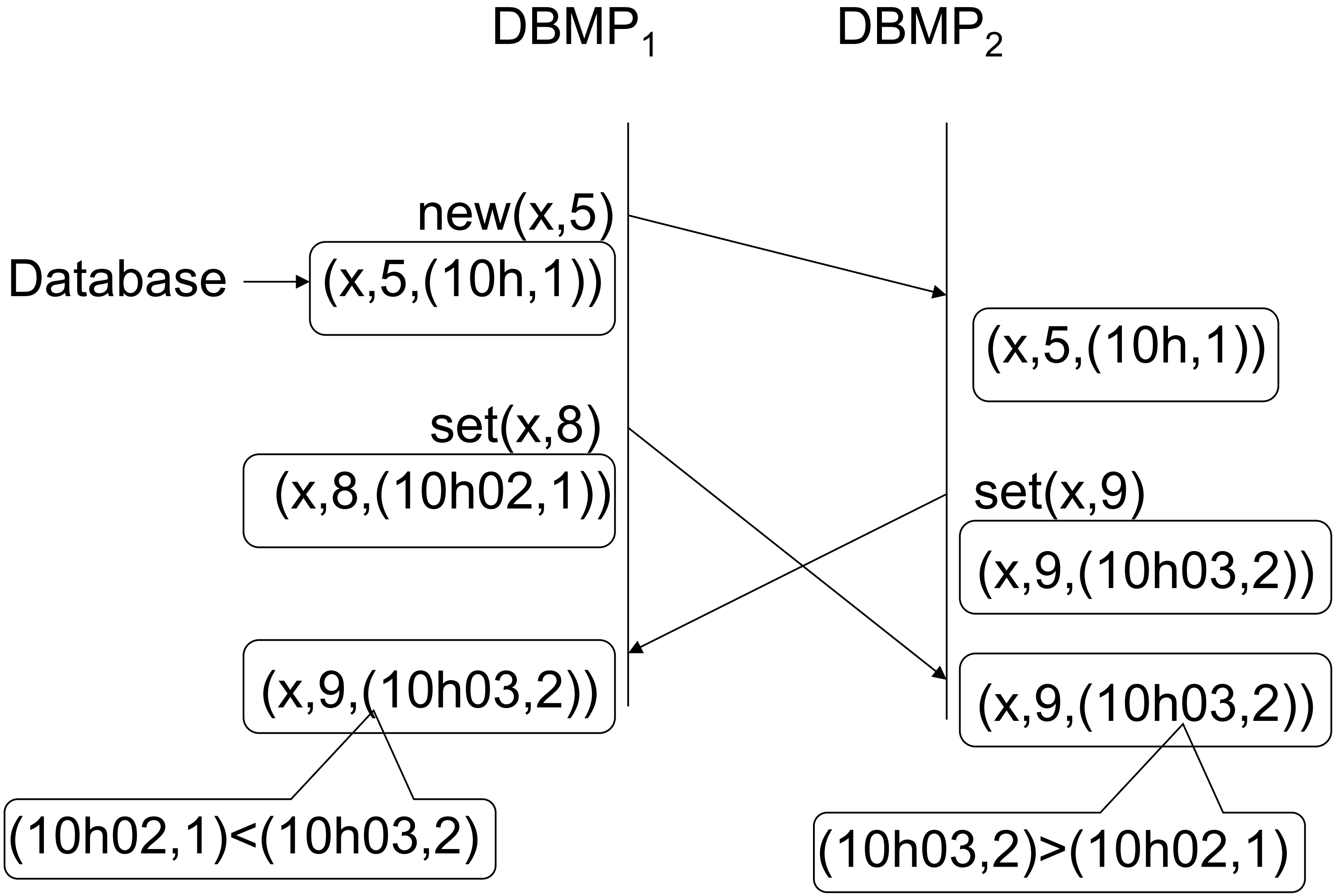
Thomas Timestamps

- In the face of concurrent modifications to an entry, how to select the « most recent » change?
- Thomas timestamps before Lamport timestamps !
- A timestamp is a pair (T,D)
    - T is a network time standard (time-of-day)
    - D is a DBMP identifier
- Timestamps comparison
    - (T1,D1)>(T2,D2) iff (T1>T2) or (T1=T2 and D1>D2)
- If D1=D2 and T1=T2, then the same operation

Database entry

- E::=(S,V,T)
  - S is the selector
  - V is the value
  - T is the timestamp = (Time, DBMP id) of the last change to the entry

Thomas write rule = last writer wins



DBMP$_1$

DBMP$_2$

new(x,5)

Database → (x,5,(10h,1))

(x,5,(10h,1))

set(x,8)

(x,8,(10h02,1))

set(x,9)

(x,9,(10h03,2))

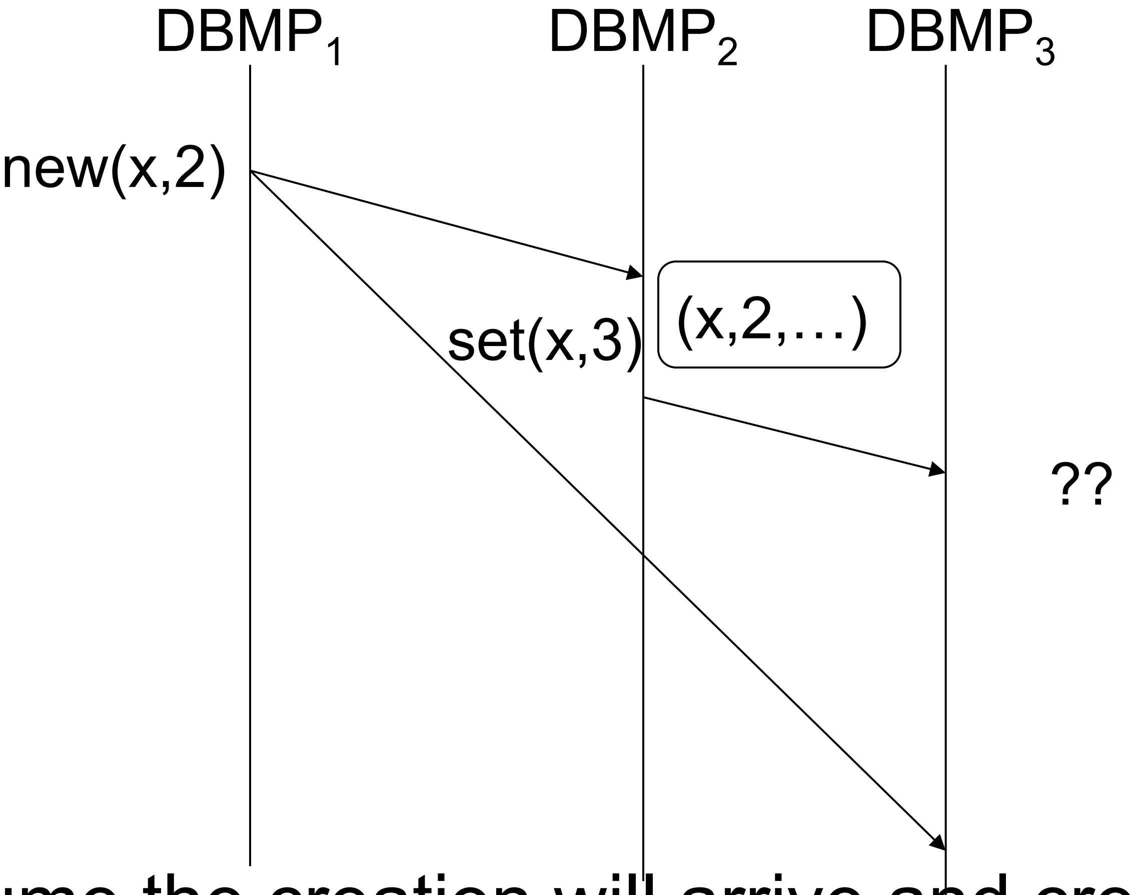
(x,9,(10h03,2))

(x,9,(10h03,2))

(10h02,1)<(10h03,2)

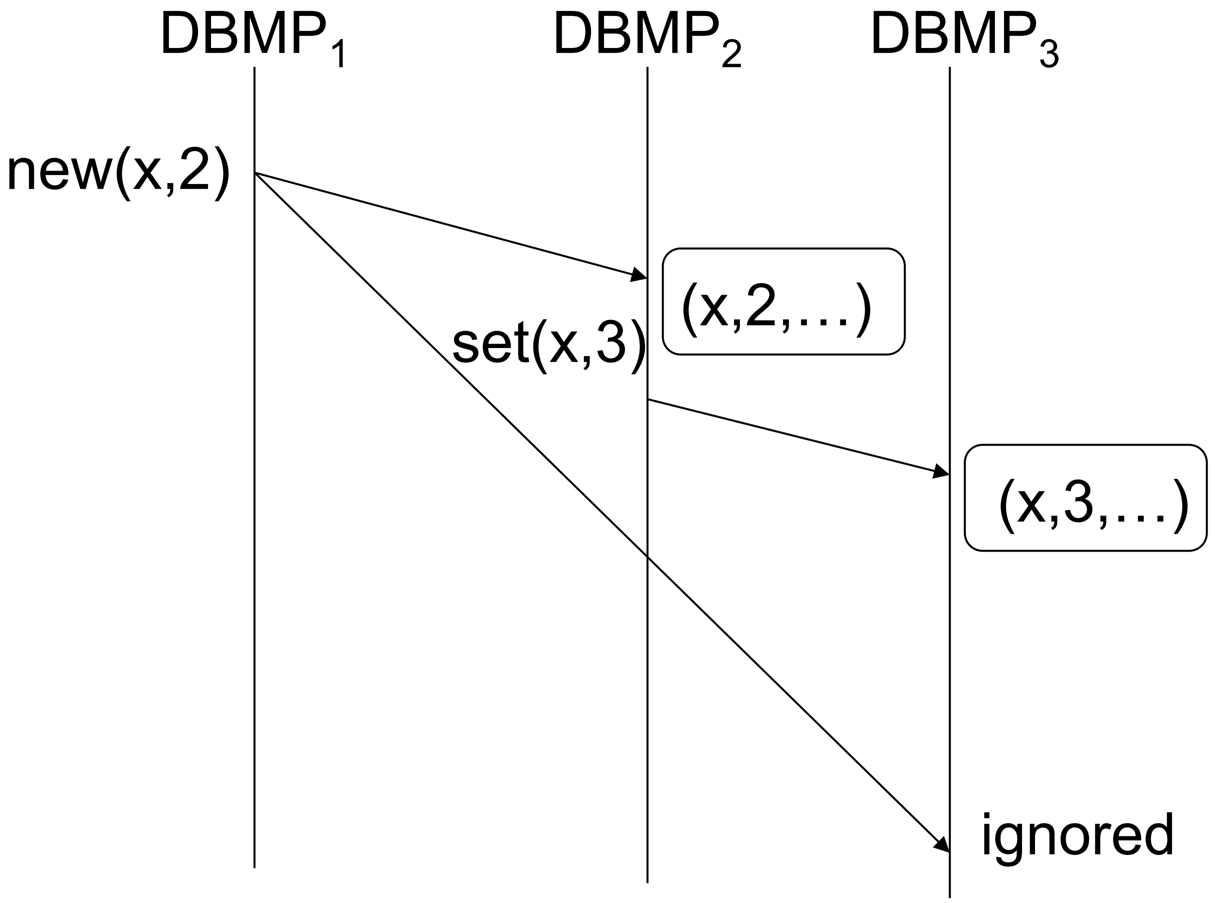
(10h03,2)>(10h02,1)

Creation/update

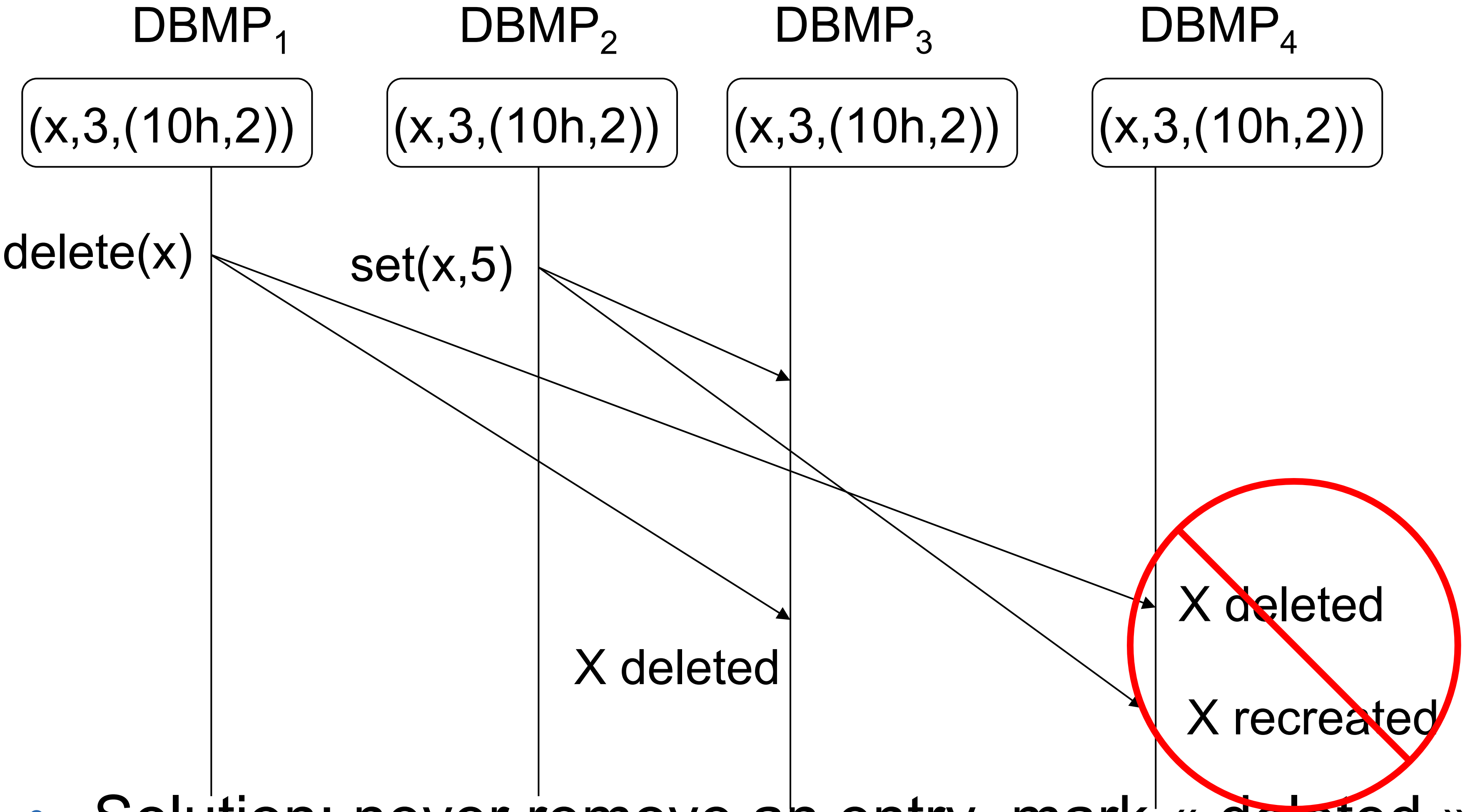


- Assume the creation will arrive and create the entry right away
- Creation operation ignored at arrival

Creation/update



Deletion



DBMP_1 DBMP_2 DBMP_3 DBMP_4

(x,3,(10h,2)) (x,3,(10h,2)) (x,3,(10h,2)) (x,3,(10h,2))

delete(x)

set(x,5)

X deleted

X deleted

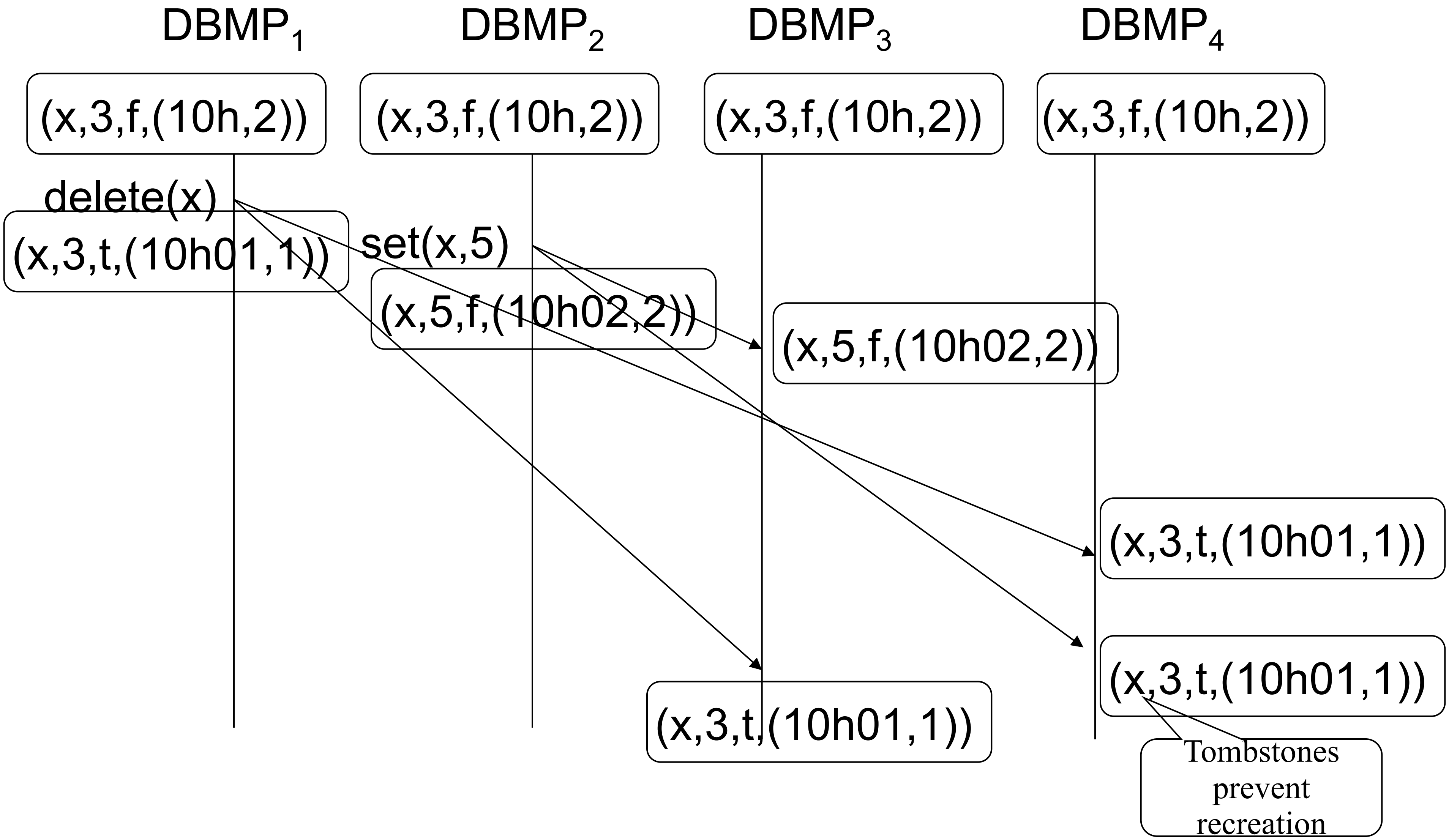
X recreated

- Solution: never remove an entry, mark « deleted » flag

Tombstones

- E::=(S,V,F,T)
    - S is the selector
    - V is the value
    - F is the deleted/not-deleted flag
    - T is the timestamp = (Time, DBMP id) of the last change to the entry
- F=t if deleted
- F=f if not-deleted

Tombstones



$DBMP_1$ $DBMP_2$ $DBMP_3$ $DBMP_4$

(x,3,f,(10h,2)) (x,3,f,(10h,2)) (x,3,f,(10h,2)) (x,3,f,(10h,2))

delete(x)

(x,3,t,(10h01,1))

set(x,5)

(x,5,f,(10h02,2))

(x,5,f,(10h02,2))

(x,3,t,(10h01,1))

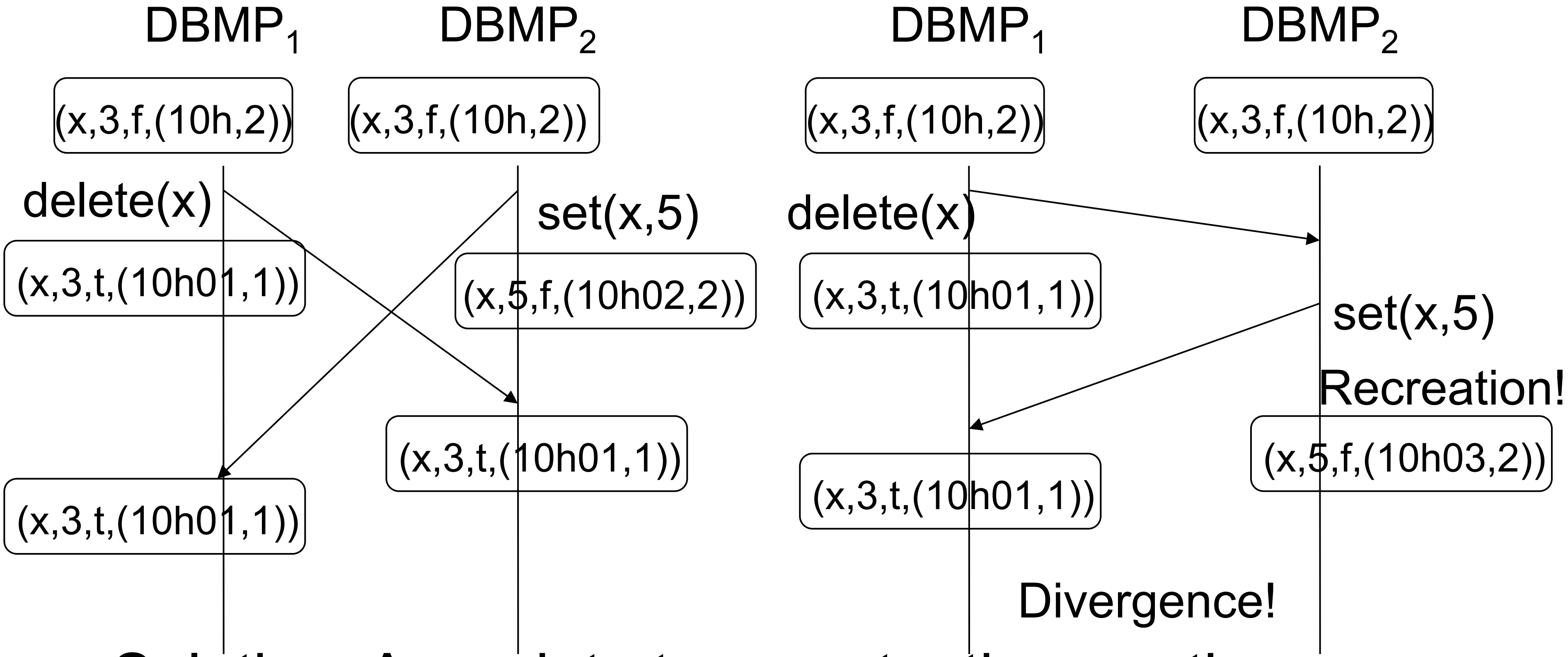
(x,3,t,(10h01,1))

(x,3,t,(10h01,1))

Tombstones prevent recreation

Tombstones

- DBMP1 cannot distinguish in which of the two cases DBMP2 is



$DBMP_1$ $DBMP_2$

(x,3,f,(10h,2)) (x,3,f,(10h,2))

delete(x)

(x,3,t,(10h01,1))

set(x,5)

(x,5,f,(10h02,2))

(x,3,t,(10h01,1))

(x,3,t,(10h01,1))

$DBMP_1$ $DBMP_2$

(x,3,f,(10h,2)) (x,3,f,(10h,2))

delete(x)

(x,3,t,(10h01,1))

set(x,5)

Recreation!

(x,5,f,(10h03,2))

(x,3,t,(10h01,1))

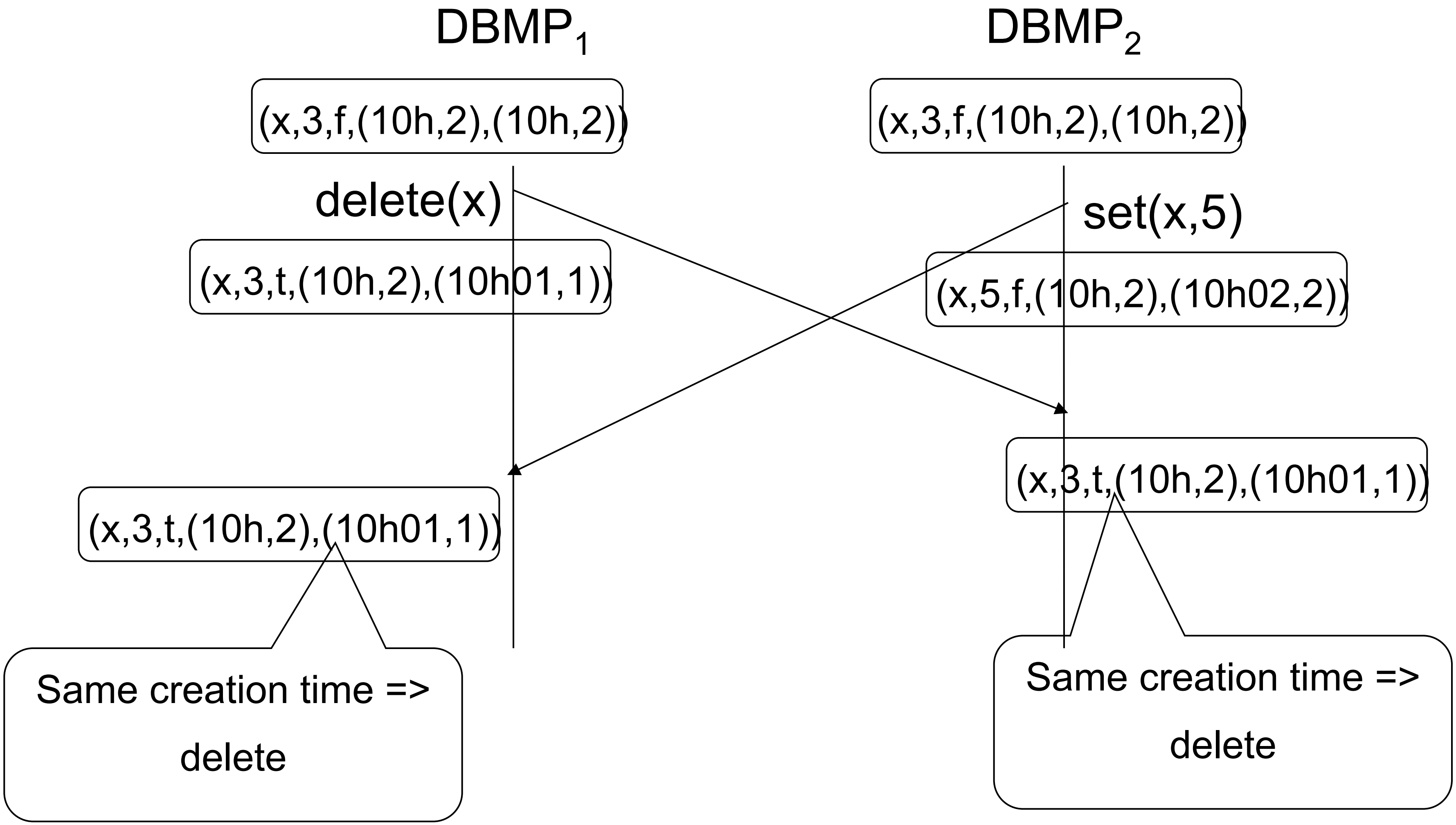
Divergence!

- Solution: Associate to an entry the creation timestamp

Tombstones

- E::=(S,V,F,CT,T)
  - S is the selector
  - V is the value
  - F is the deleted/not-deleted flag
  - CT is the timestamp for creation
  - T is the timestamp = (Time, DBMP id) of the last change to the entry
- If F=f and CT=T, then creation
- If F=f and CT<T, then assignment
- If F=t, then deletion

Tombstones



DBMP$_1$

DBMP$_2$

(x,3,f,(10h,2),(10h,2))

(x,3,f,(10h,2),(10h,2))

delete(x)

set(x,5)

(x,3,t,(10h,2),(10h01,1))

(x,5,f,(10h,2),(10h02,2))

(x,3,t,(10h,2),(10h01,1))

(x,3,t,(10h,2),(10h01,1))

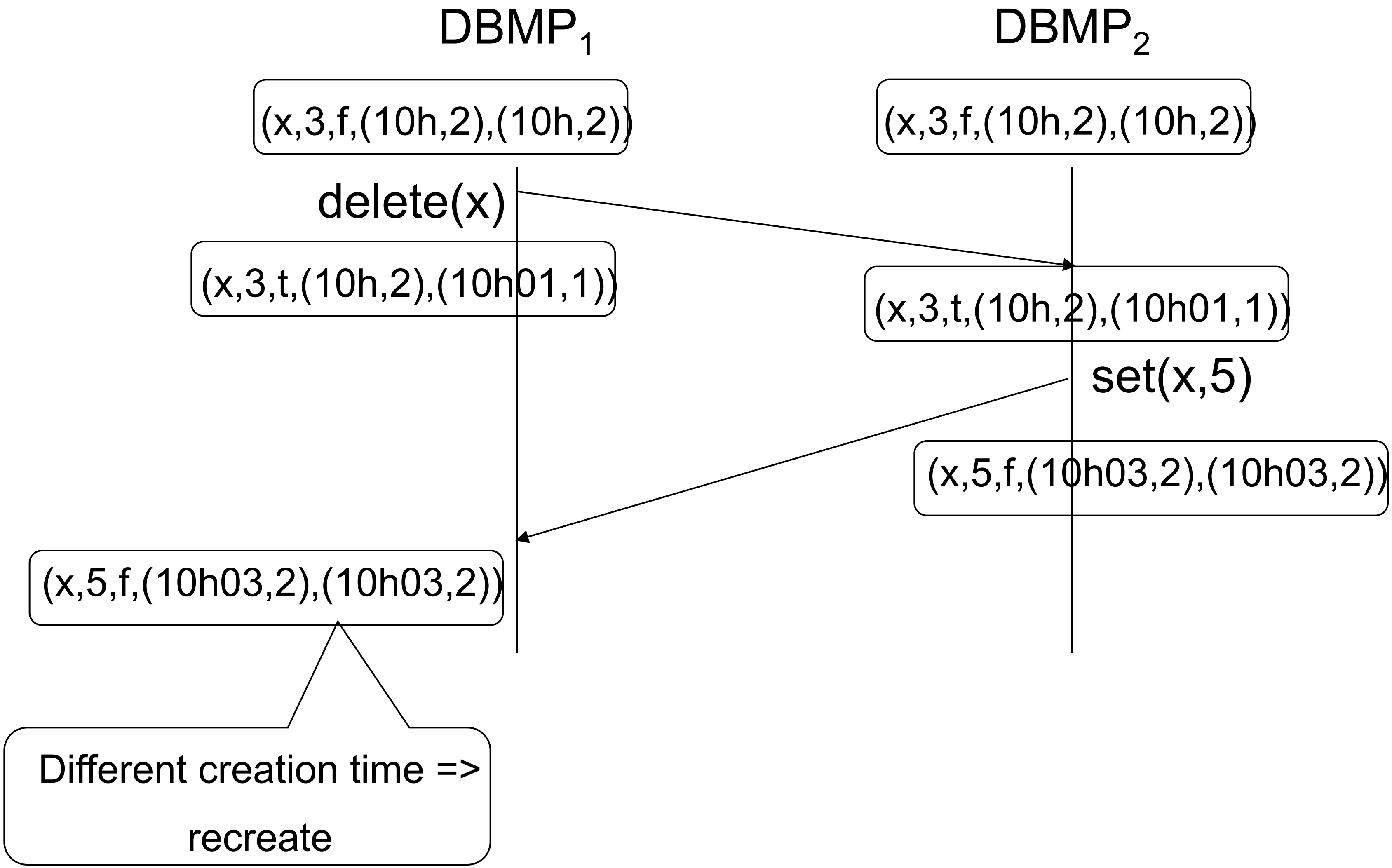
Same creation time =>

delete

Same creation time =>

delete

Tombstones



$DBMP_1$

$DBMP_2$

(x,3,f,(10h,2),(10h,2))

(x,3,f,(10h,2),(10h,2))

delete(x)

(x,3,t,(10h,2),(10h01,1))

(x,3,t,(10h,2),(10h01,1))

set(x,5)

(x,5,f,(10h03,2),(10h03,2))

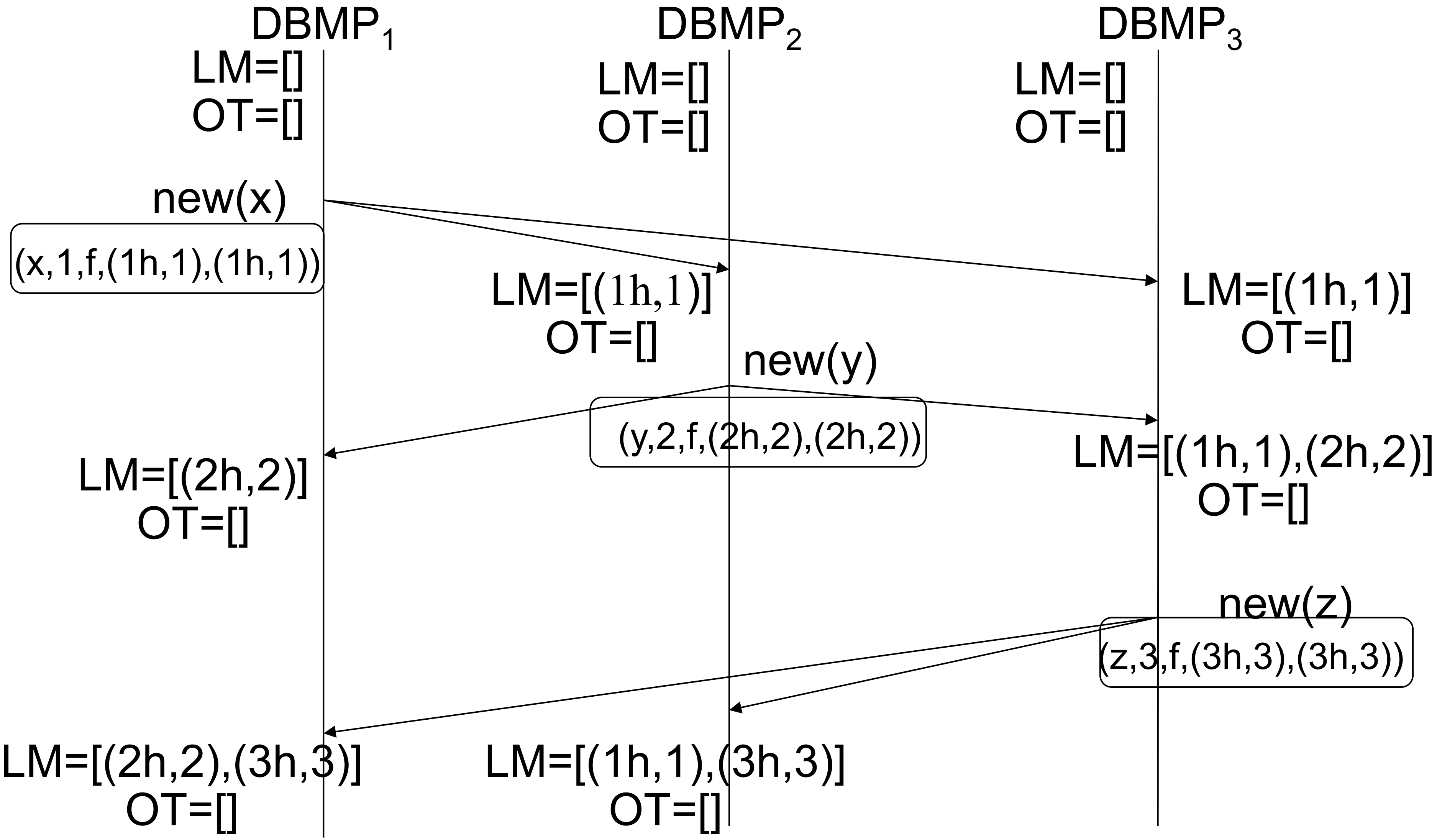
(x,5,f,(10h03,2),(10h03,2))

Different creation time => recreate

Garbage collection (of deleted elt)

- Make sure of no reception of assignments with same S and the same or older CT
- Remember assumption: Modifications of a DBMP delivered in sequential order
- Each DBMP maintains two « timestamp vectors »
  - Last modifications from all DBMPs
    - LM[i] last timestamp from DBMP i
    - Modified each time an operation is received
  - Oldest timestamps received by each DBMP
    - OT[i] oldest timestamp received by DBMP i
    - Sent upon reception of a delete
- Can do garbage collection if timestamp of delete <= timestamp of min(OT)

Garbage collection



DBMP$_1$

LM=[]
OT=[]

new(x)

(x,1,f,(1h,1),(1h,1))

LM=[(2h,2)]
OT=[]

LM=[(2h,2),(3h,3)]
OT=[]

DBMP$_2$

LM=[]
OT=[]

LM=[(1h,1)]
OT=[]

new(y)

(y,2,f,(2h,2),(2h,2))

LM=[(1h,1),(3h,3)]
OT=[]

DBMP$_3$

LM=[]
OT=[]

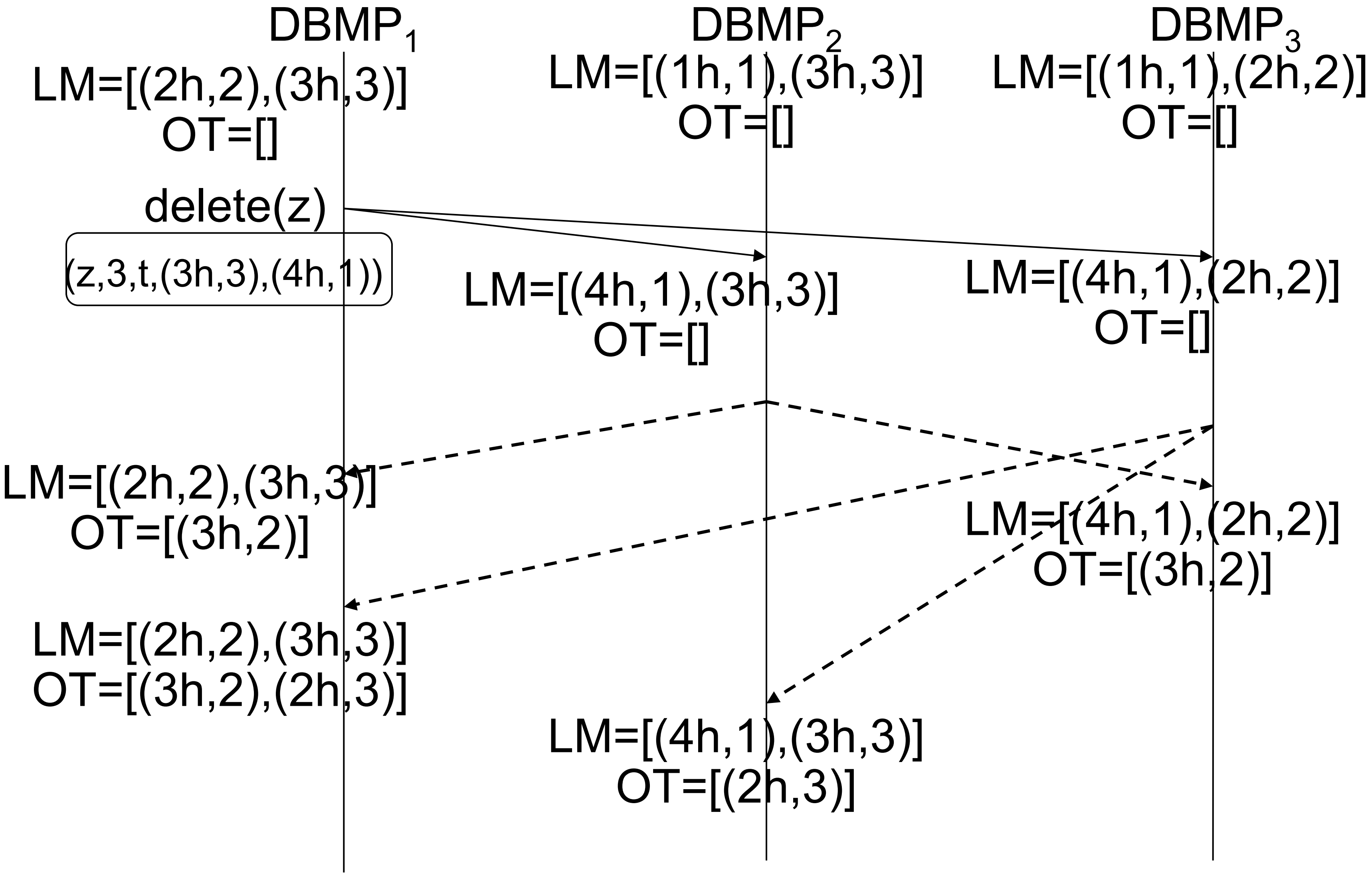
LM=[(1h,1)]
OT=[]

LM=[(1h,1),(2h,2)]
OT=[]

new(z)

(z,3,f,(3h,3),(3h,3))

Garbage collection



$DBMP_1$

LM=[(2h,2),(3h,3)]
OT=[]

$DBMP_2$

LM=[(1h,1),(3h,3)]
OT=[]

$DBMP_3$

LM=[(1h,1),(2h,2)]
OT=[]

delete(z)

(z,3,t,(3h,3),(4h,1))

LM=[(4h,1),(3h,3)]
OT=[]

LM=[(4h,1),(2h,2)]
OT=[]

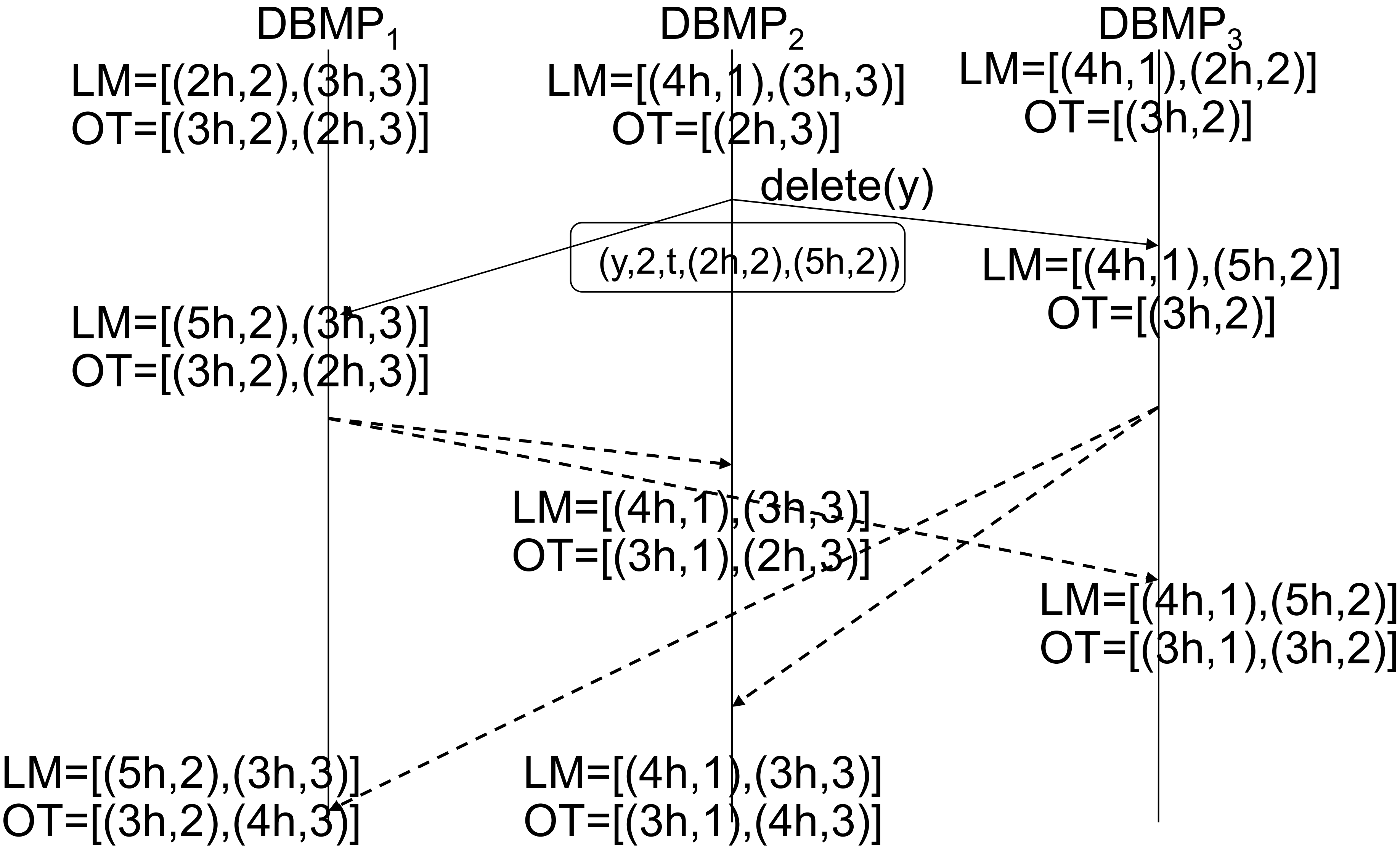
LM=[(2h,2),(3h,3)]
OT=[(3h,2)]

LM=[(4h,1),(2h,2)]
OT=[(3h,2)]

LM=[(2h,2),(3h,3)]
OT=[(3h,2),(2h,3)]

LM=[(4h,1),(3h,3)]
OT=[(2h,3)]

Garbage collection



DBMP$_1$

LM=[(2h,2),(3h,3)]
OT=[(3h,2),(2h,3)]

DBMP$_2$

LM=[(4h,1),(3h,3)]
OT=[(2h,3)]

DBMP$_3$

LM=[(4h,1),(2h,2)]
OT=[(3h,2)]

delete(y)

(y,2,t,(2h,2),(5h,2))

LM=[(4h,1),(5h,2)]
OT=[(3h,2)]

LM=[(5h,2),(3h,3)]
OT=[(3h,2),(2h,3)]

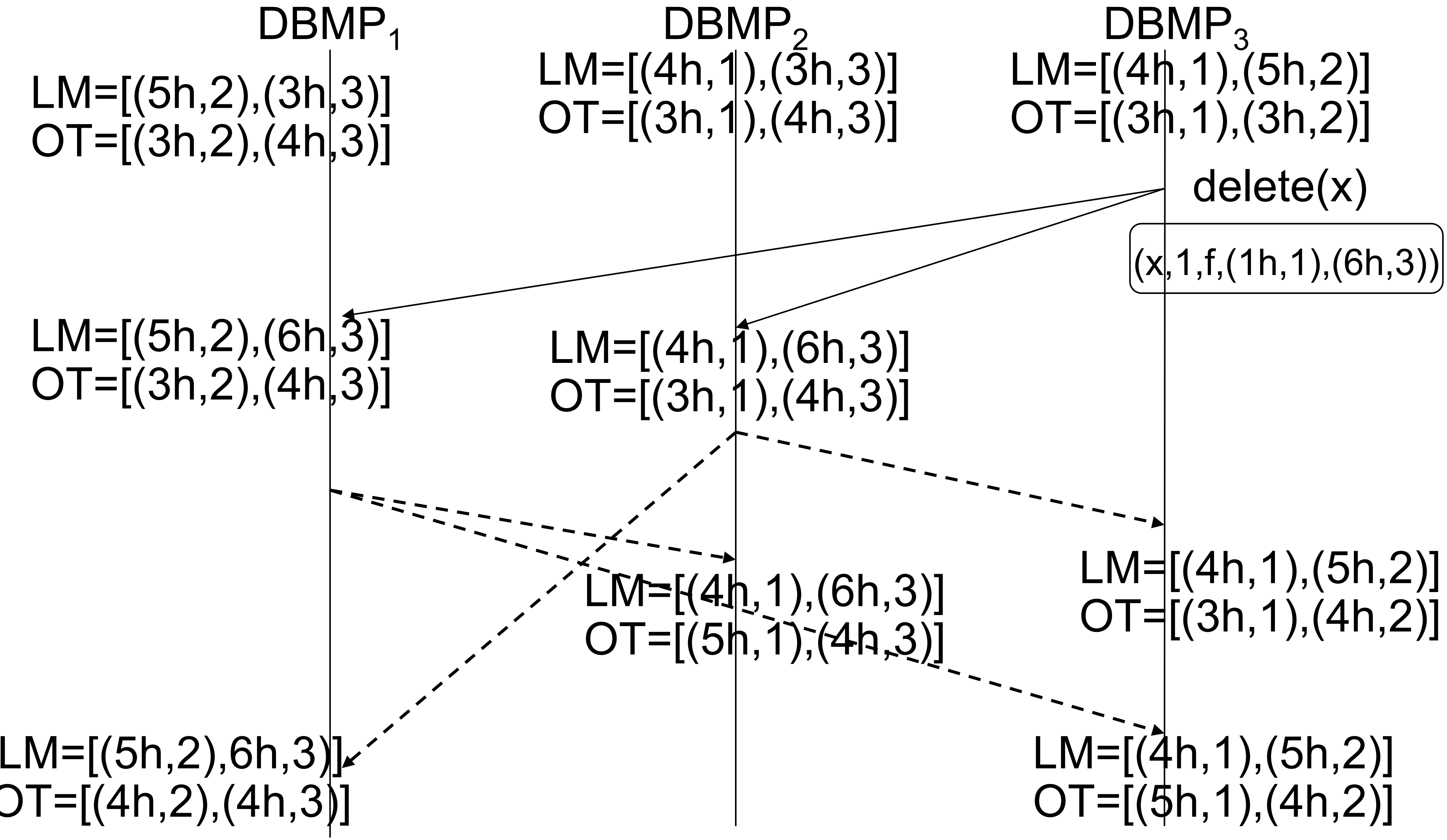
LM=[(4h,1),(3h,3)]
OT=[(3h,1),(2h,3)]

LM=[(4h,1),(5h,2)]
OT=[(3h,1),(3h,2)]

LM=[(5h,2),(3h,3)]
OT=[(3h,2),(4h,3)]

LM=[(4h,1),(3h,3)]
OT=[(3h,1),(4h,3)]

Garbage collection



DBMP$_1$

LM=[(5h,2),(3h,3)]
OT=[(3h,2),(4h,3)]

LM=[(5h,2),(6h,3)]
OT=[(3h,2),(4h,3)]

LM=[(5h,2),6h,3)]
OT=[(4h,2),(4h,3)]

DBMP$_2$

LM=[(4h,1),(3h,3)]
OT=[(3h,1),(4h,3)]

LM=[(4h,1),(6h,3)]
OT=[(3h,1),(4h,3)]

LM=[(4h,1),(6h,3)]
OT=[(5h,1),(4h,3)]

DBMP$_3$

LM=[(4h,1),(5h,2)]
OT=[(3h,1),(3h,2)]

delete(x)

(x,1,f,(1h,1),(6h,3))

LM=[(4h,1),(5h,2)]
OT=[(3h,1),(4h,2)]

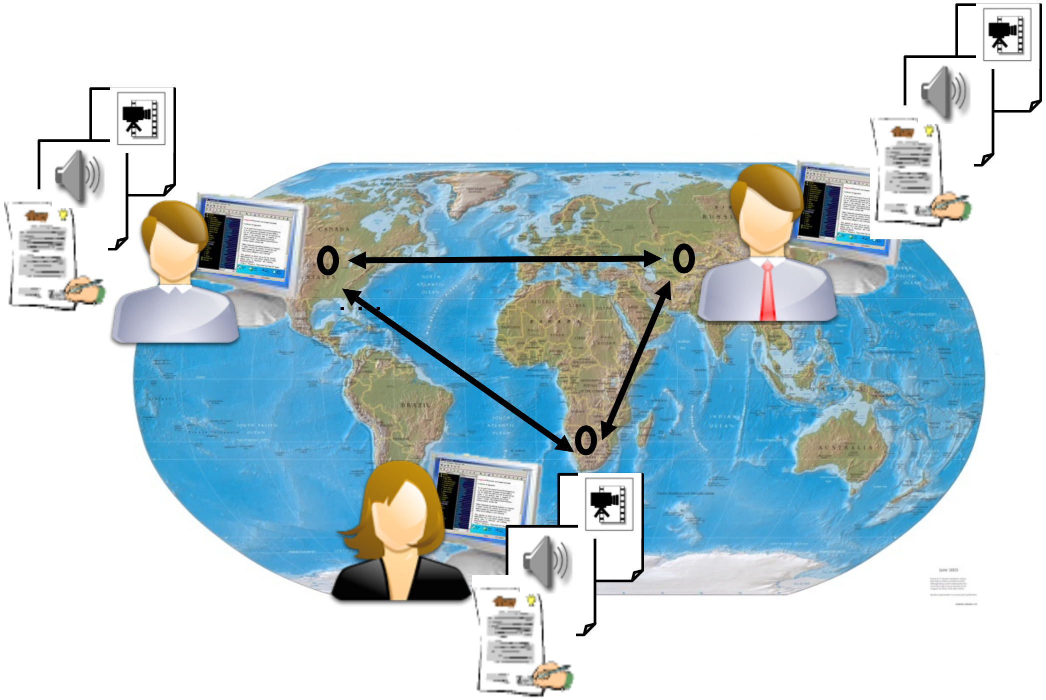
LM=[(4h,1),(5h,2)]
OT=[(5h,1),(4h,2)]

- z can be garbaged

Agenda

- Optimistic replication approaches
  - Operational transformation
    - General ideas
    - Transformation functions
      - Properties to be ensured
      - Examples
    - Integration algorithms
      - SOCT2
      - Other algorithms next lecture

Operational transformation



- Domain of application: collaborative editing
- Document replication
  - Disconnected work
  - Better response time for real-time collaboration

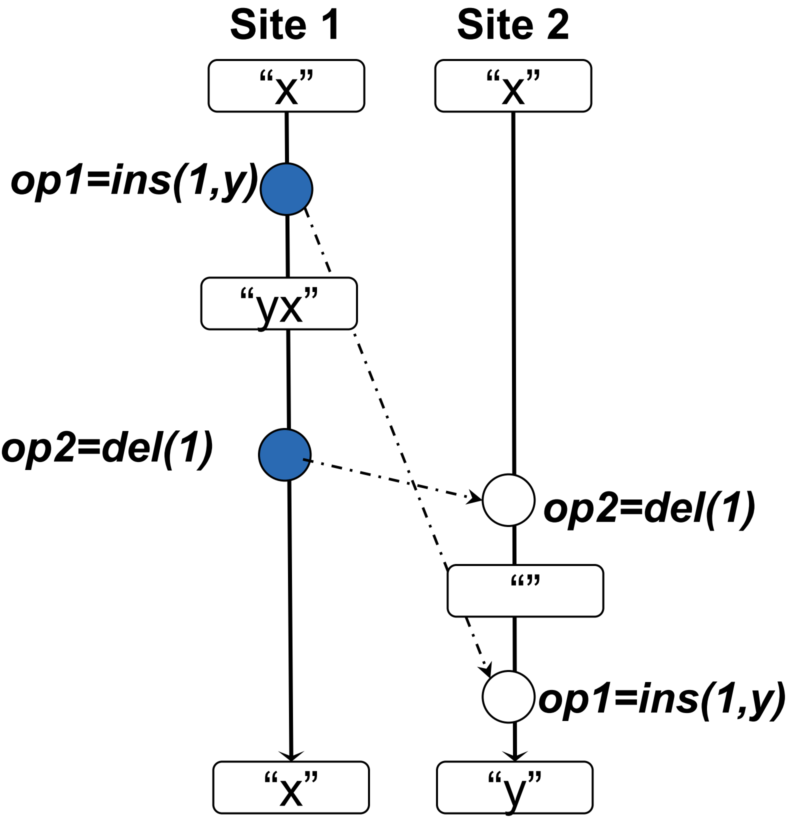
Operational transformation

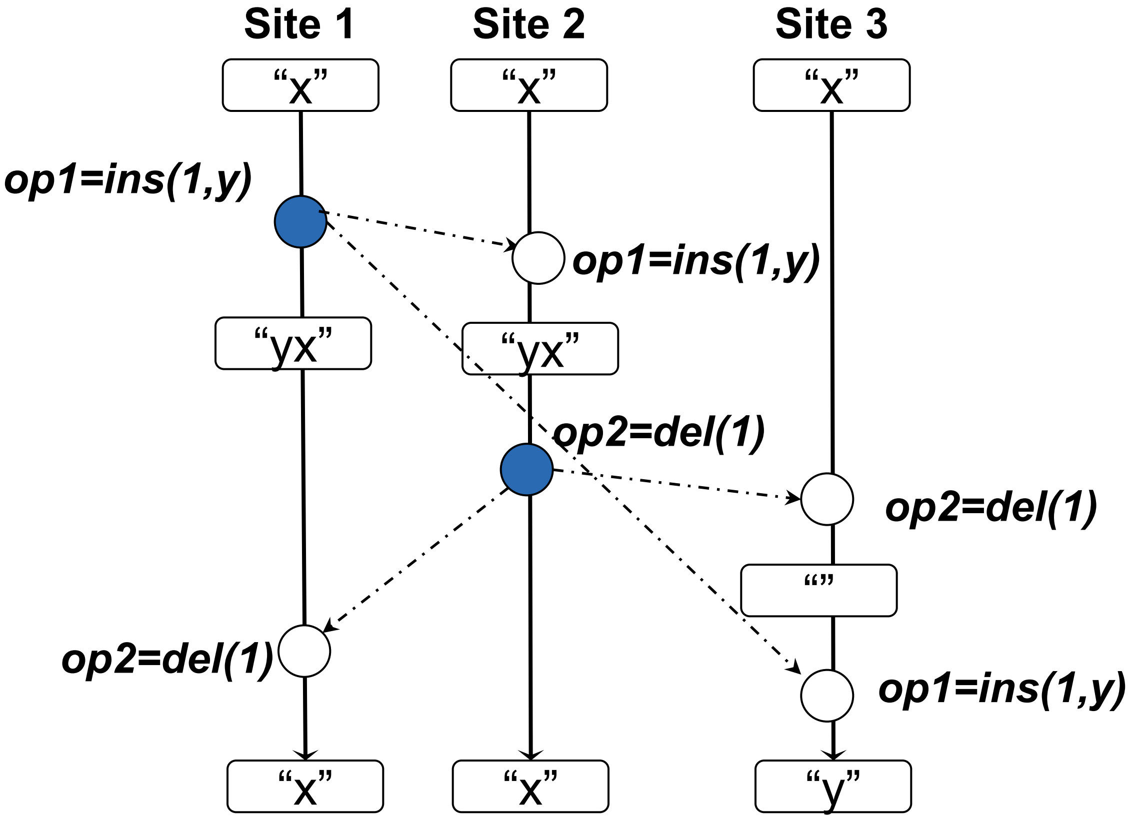
- Optimistic replication model
  - An operation is :
    - Locally executed,
    - Sent to other sites,
    - Received by a site,
    - Transformed according to concurrent operations,
    - Executed on local copy
- 2 components :
  - An integration algorithm : diffusion, integration
  - Some transformation functions

Operational transformation

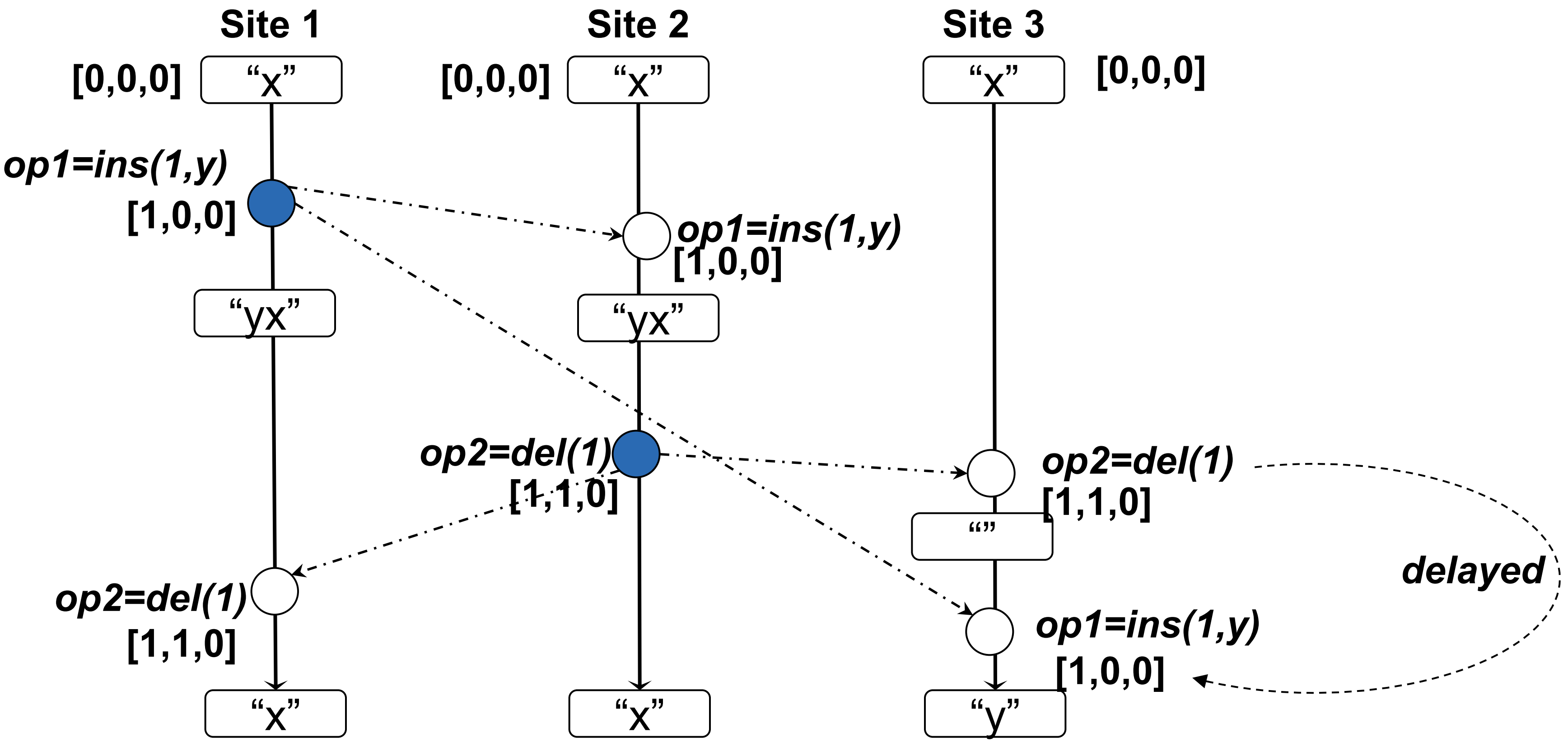
- Textual documents seen as a sequence of characters
- Operations
  - ins(p,c)
  - del(p)
- Three main issues
  - Causality preservation
  - Intention preservation
  - Convergence

Causality


Site 1
Site 2
"x"
"x"
op1=ins(1,y)
"yx"
op2=del(1)
op2=del(1)
""
op1=ins(1,y)
"x"
"y"


Site 1
Site 2
Site 3
"x"
"x"
"x"
op1=ins(1,y)
op1=ins(1,y)
"yx"
"yx"
op2=del(1)
op2=del(1)
""
op2=del(1)
op1=ins(1,y)
"x"
"x"
"y"

Causality


Site 1
Site 2
Site 3
[0,0,0]
"x"
[0,0,0]
"x"
"x"
[0,0,0]
op1=ins(1,y)
[1,0,0]
op1=ins(1,y)
[1,0,0]
"yx"
"yx"
op2=del(1)
[1,1,0]
op2=del(1)
[1,1,0]
""
delayed
op2=del(1)
[1,1,0]
op1=ins(1,y)
[1,0,0]
"x"
"x"
"y"

Intention

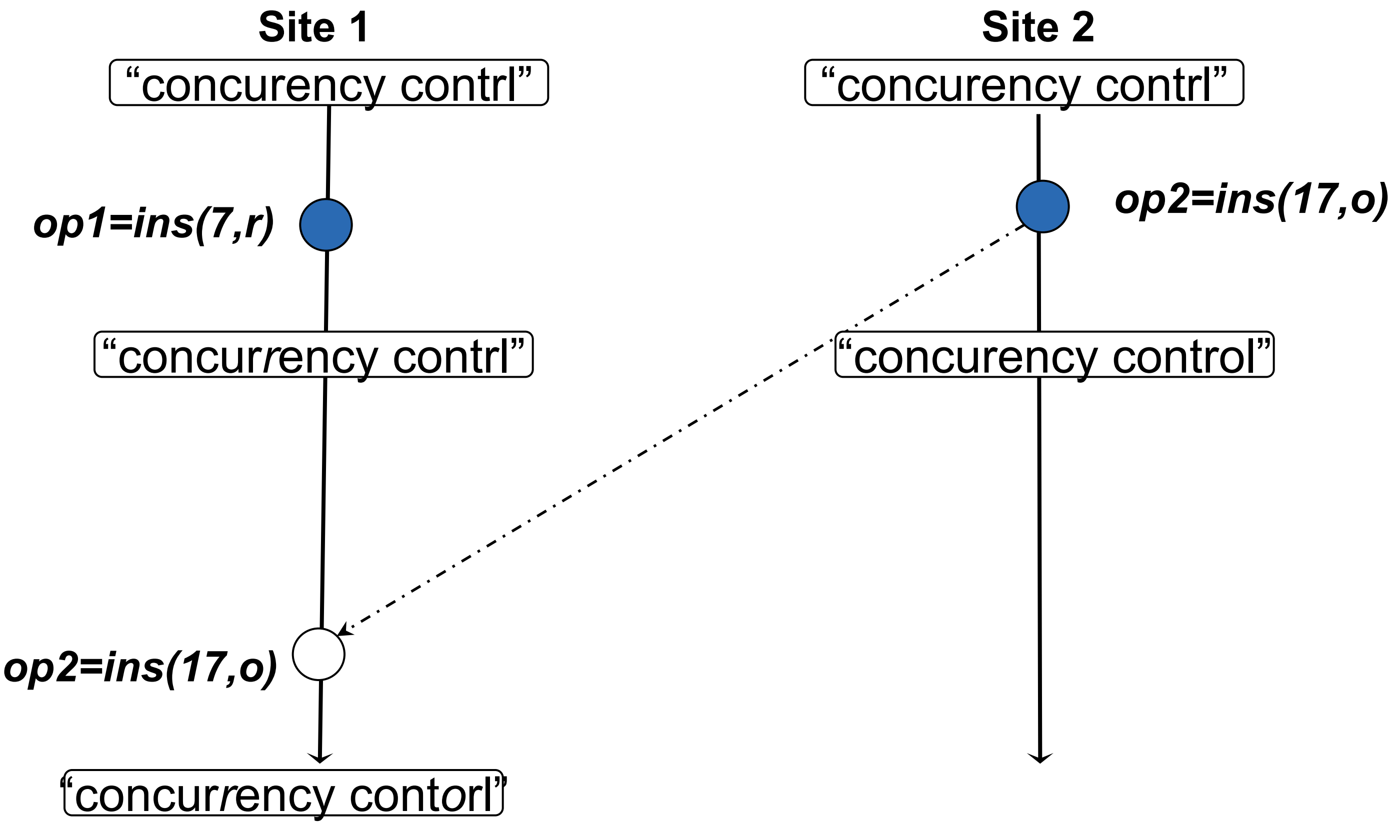
- Intention of an operation is the observed effect as result of its execution on its generation state
- Passing from initial state “ab” to final state “aXb” we can observe:
  - ins(2,X)
  - ins(a<X<b)
  - ins(a<X)
  - ins(X>b)

Preserving user intention (*)

- 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 *O* does not change the effects of independent operations.

(*) Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63−108, March 1998.

Intention violation



Site 1

“concurency contrl”

op1=ins(7,r)

“concurr*r*ency contrl”

op2=ins(17,o)

“concurr*r*ency contorl”

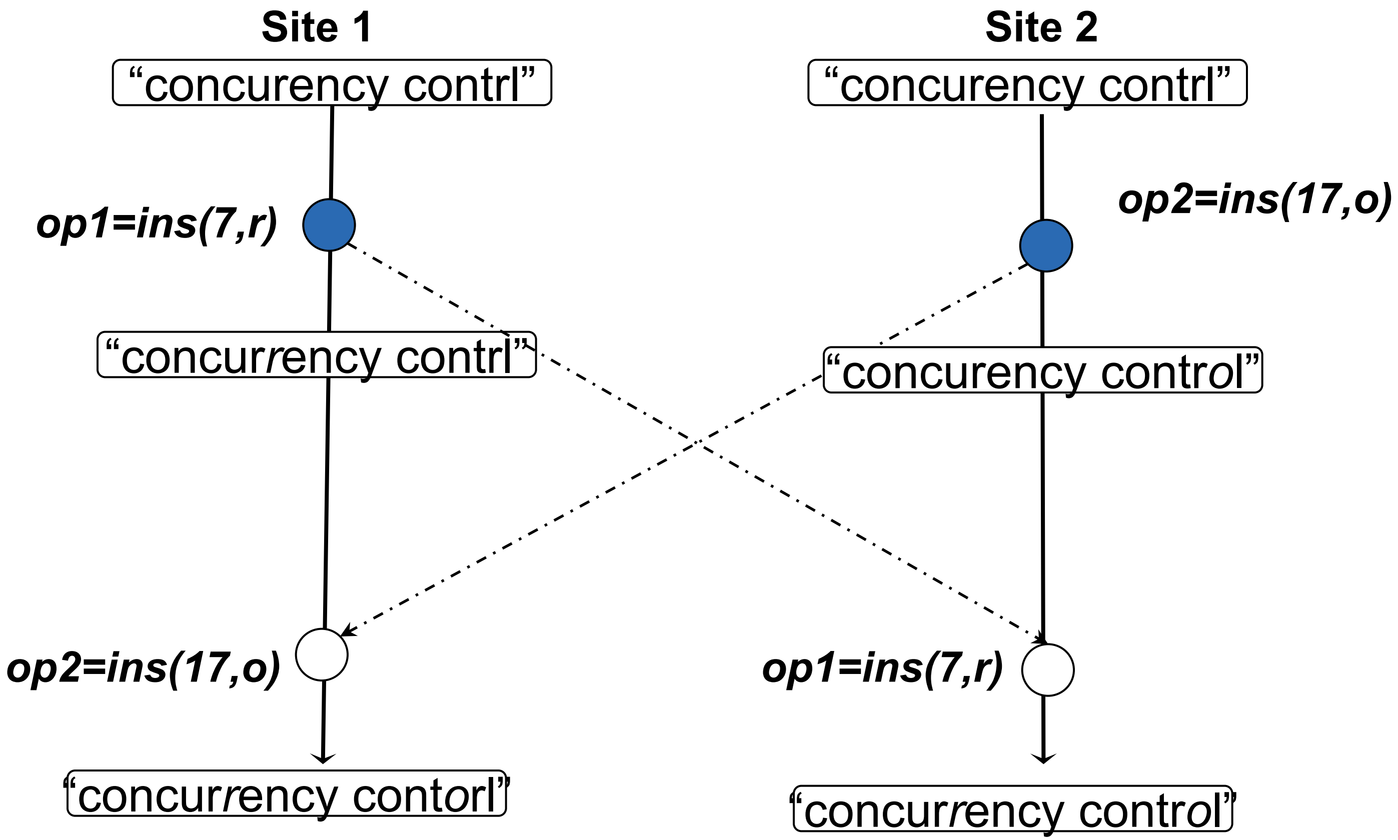
Site 2

“concurency contrl”

op2=ins(17,o)

“concurency control”

Intention violation + divergence



Site 1

“concurency contrl”

op1=ins(7,r)

“concurrency contrl”

op2=ins(17,o)

“concurrency contorl”

Site 2

“concurency contrl”

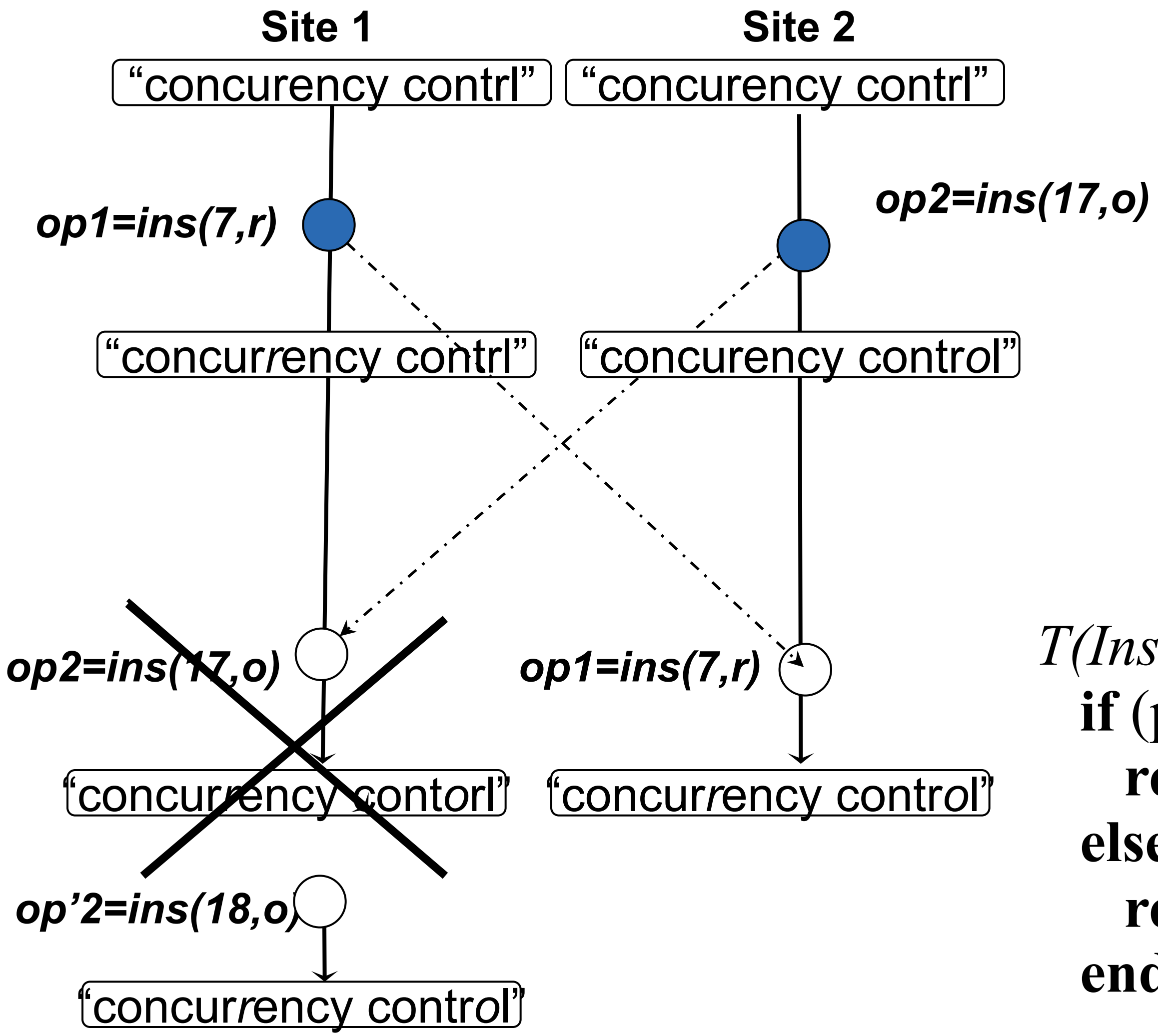
op2=ins(17,o)

“concurency control”

op1=ins(7,r)

“concurrency control”

Intention preservation



Site 1

“concurency contrl”

op1=ins(7,r)

“concurrency contrl”

op2=ins(17,o)

“concurrency contorl”

op’2=ins(18,o)

“concurrency control”

Site 2

“concurency contrl”

op2=ins(17,o)

“concurency control”

op1=ins(7,r)

“concurrency control”

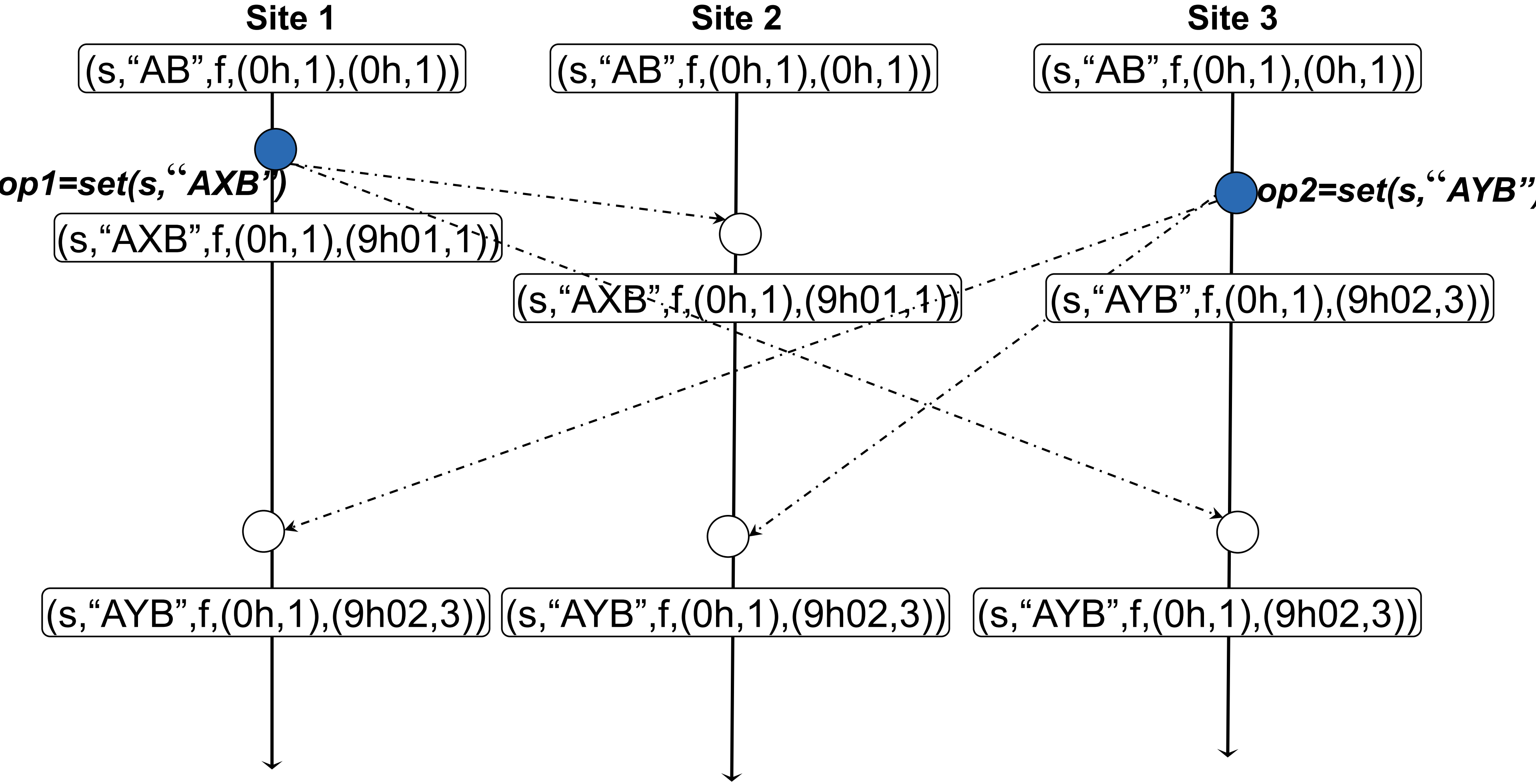
```
T(Ins(p1,c1), Ins(p2,c2)) :-
  if (p1<p2)
     return Ins(p1,c1)
  else
     return Ins(p1+1,c1)
  endif
```

Example 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()
```

Convergence but no intention preservation

Thomas Write Rule



Site 1

(s,“AB”,f,(0h,1),(0h,1))

op1=set(s,“AXB”)

(s,“AXB”,f,(0h,1),(9h01,1))

(s,“AYB”,f,(0h,1),(9h02,3))

Site 2

(s,“AB”,f,(0h,1),(0h,1))

(s,“AXB”,f,(0h,1),(9h01,1))

(s,“AYB”,f,(0h,1),(9h02,3))

Site 3

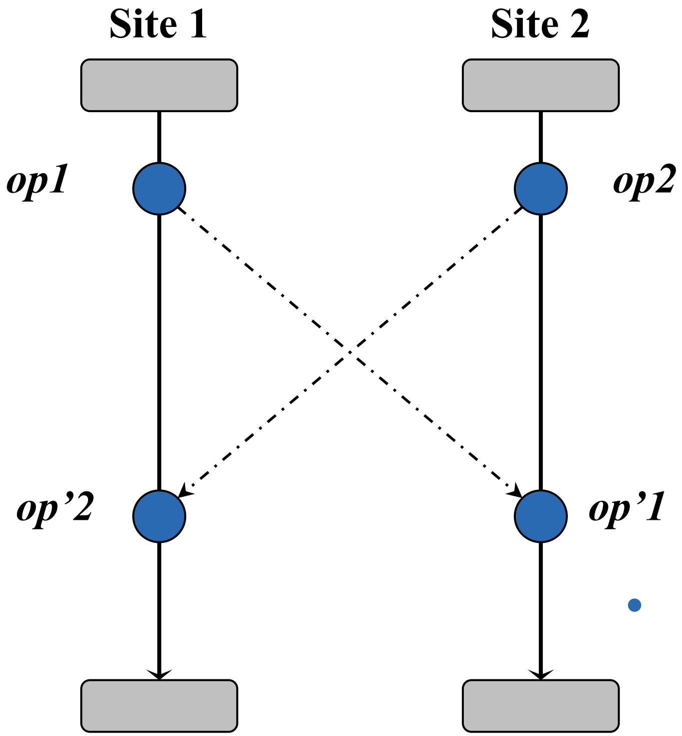
(s,“AB”,f,(0h,1),(0h,1))

op2=set(s,“AYB”

(s,“AYB”,f,(0h,1),(9h02,3))

(s,“AYB”,f,(0h,1),(9h02,3))

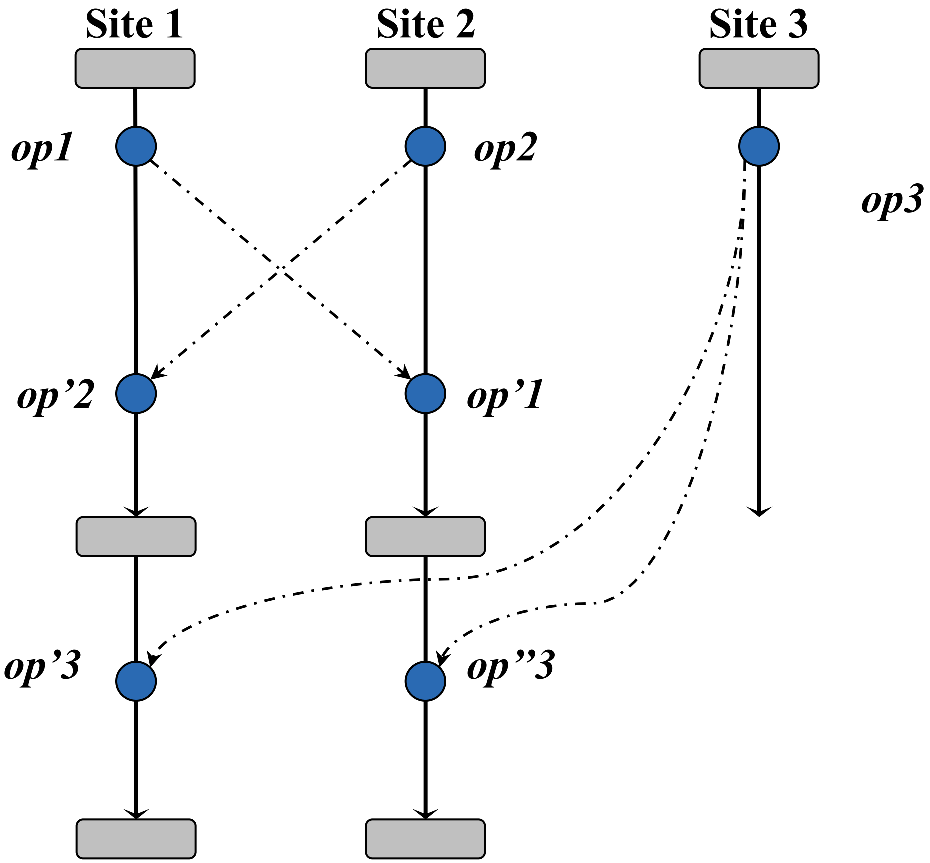
Convergence – TP1 property



- T(op2: operation, op1: operation) = op'2
  - op1 and op2 concurrent, defined on a state S
  - op'2 same effects as op2, defined on S.op1

[TP1] $op1 \circ T(op2, op1) \equiv op2 \circ T(op1,op2)$

Convergence – TP2 property

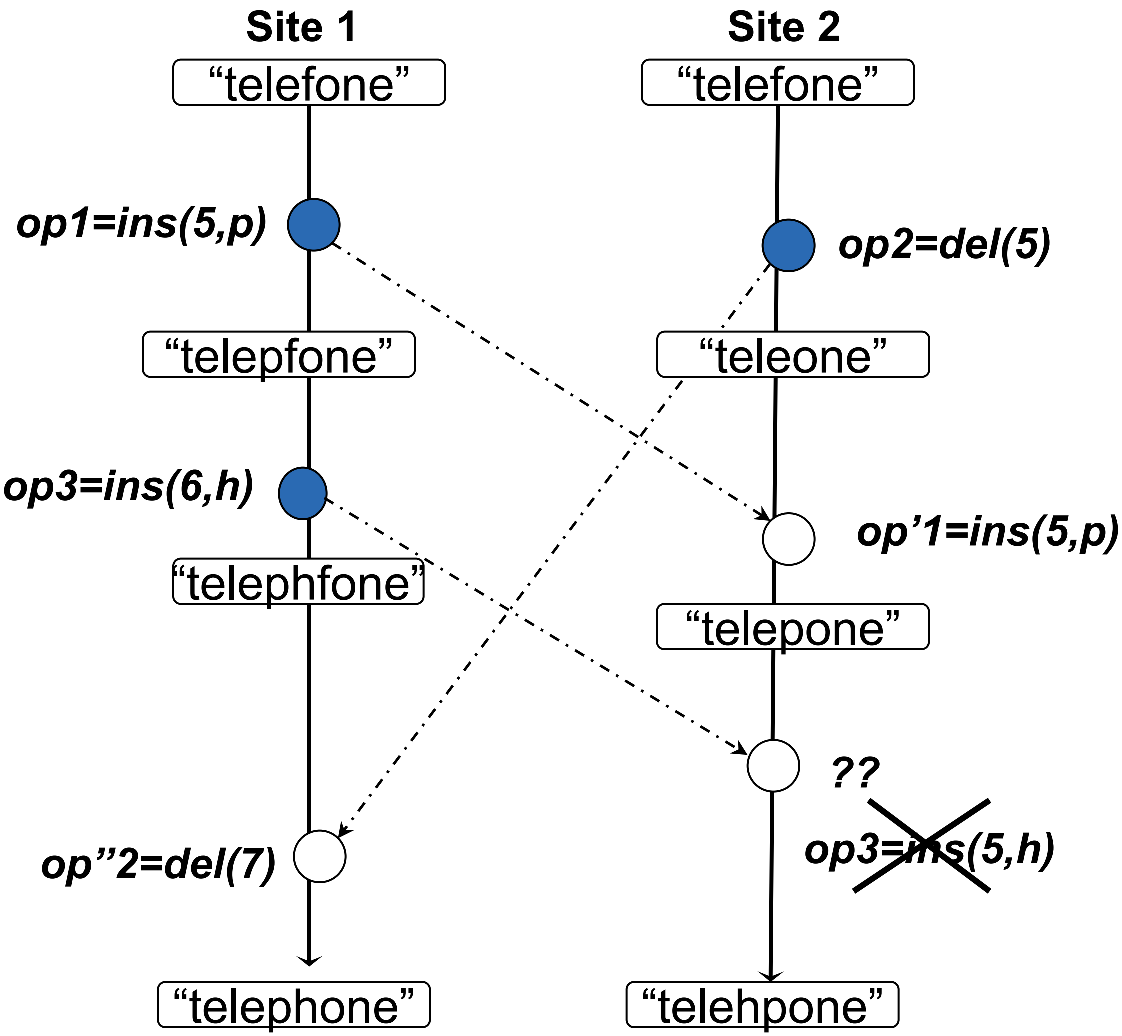


[TP2] *T(op3, op1 o T(op2, op1))= T(op3, op2 o T(op1,op2))*

OT Problems

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

Partial concurrency



Site 1

"telefone"

op1=ins(5,p)

"telepfone"

op3=ins(6,h)

"telephfone"

op''2=del(7)

"telephone"

Site 2

"telefone"

op2=del(5)

"teleone"

op'1=ins(5,p)

"telepone"

??

~~op3=ins(5,h)~~

"telehpone"

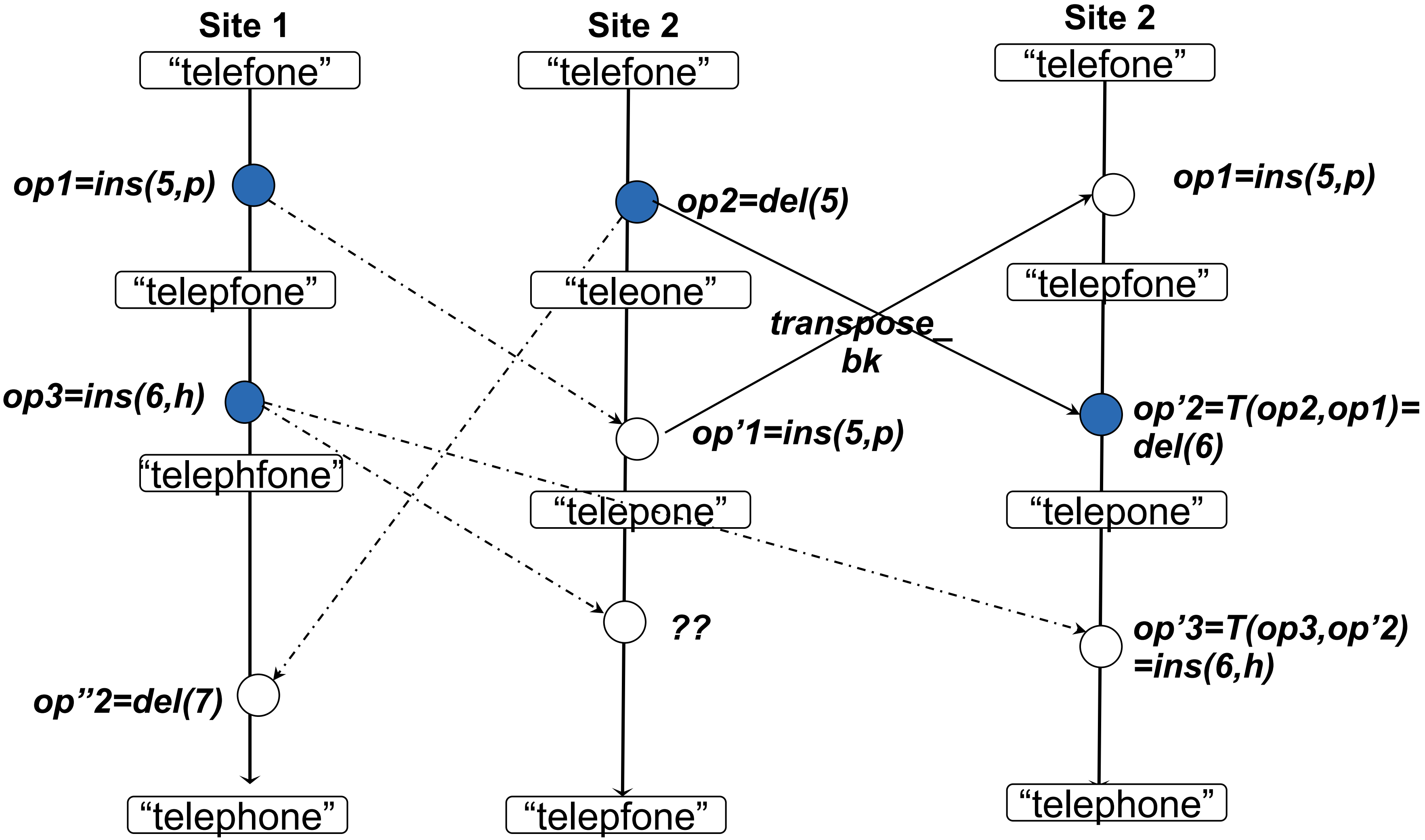
op'2=T(op2,op1)=del(6)

op''2=T(op'2,op3)=del(7)

op'1=T(op1,op2)=ins(5)

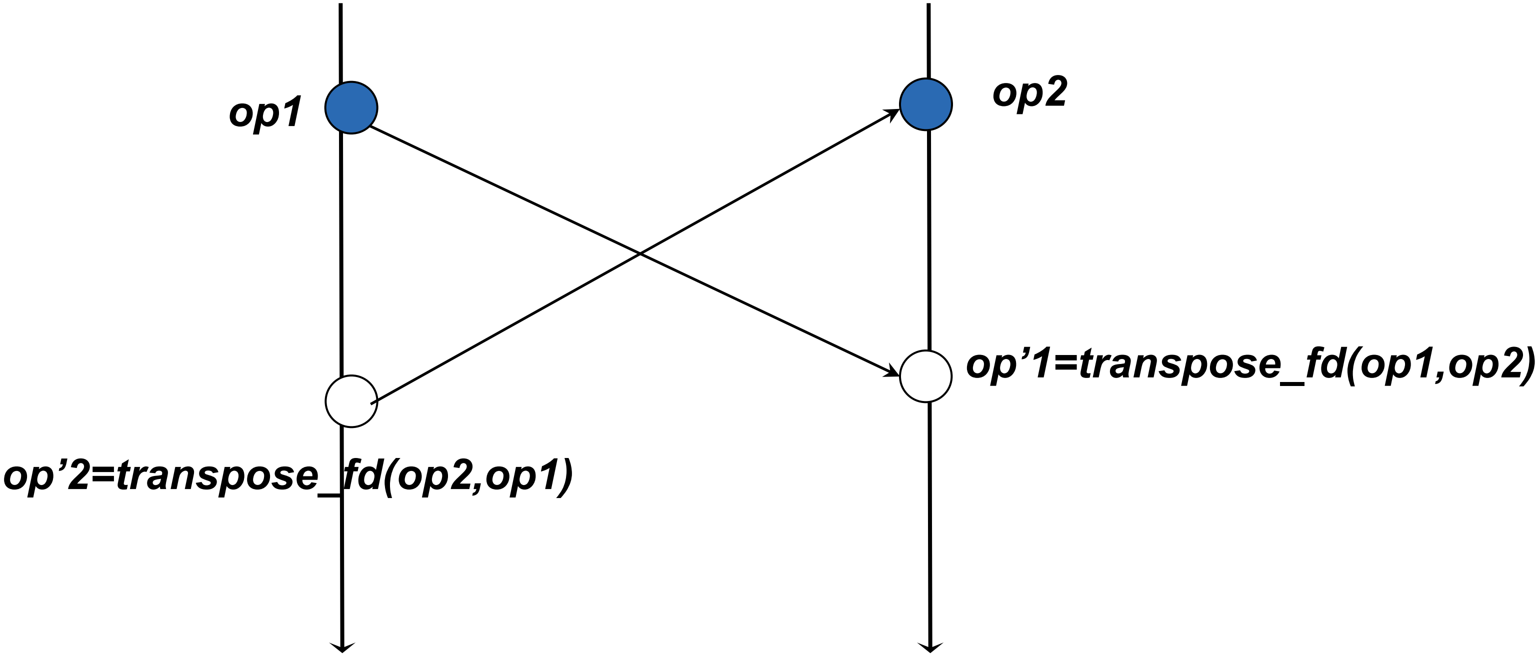
T(op3,op2) not allowed to be performed !!!

Partial concurrency



Partial concurrency

- Transpose_bk(op1,op’2)=(op2,op’1)
    - op’2=transpose_fd(op2,op1)
      Therefore op2=transpose_fd-1(op’2,op1)
    - op’1=transpose_fd(op1,op2)

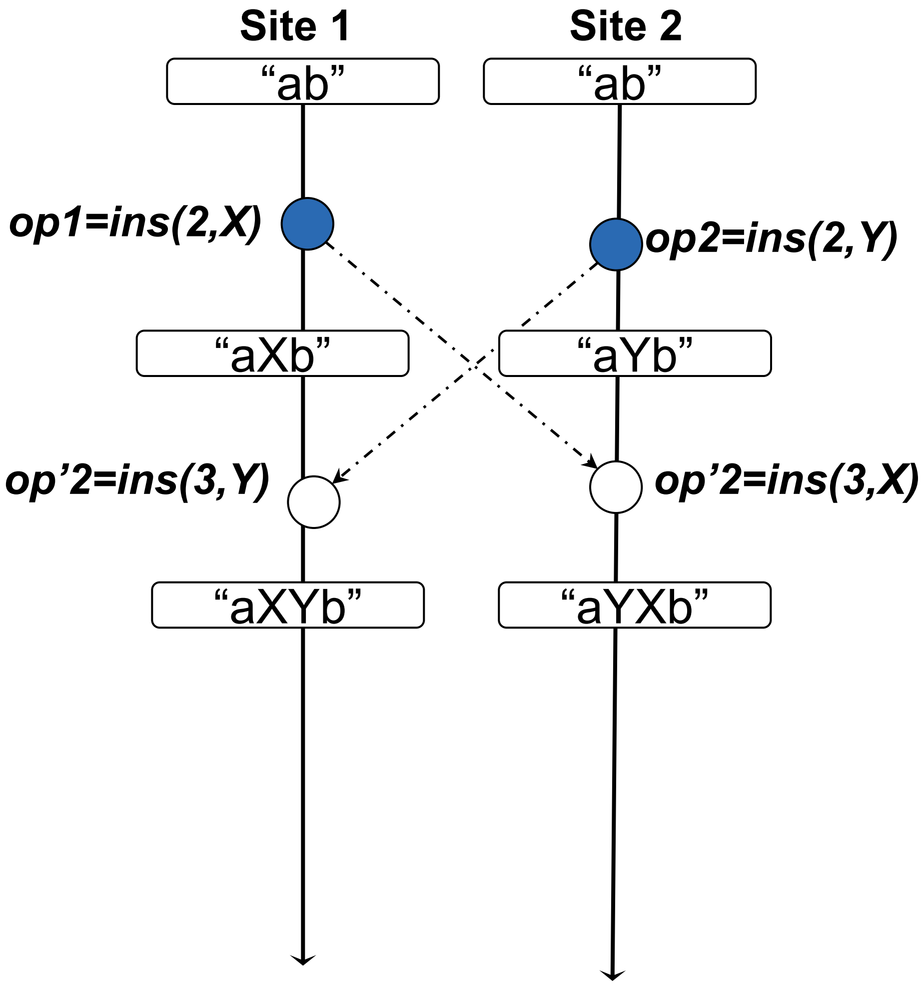


OT approaches

- Transformation functions
- Integration algorithms

Example 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()
```

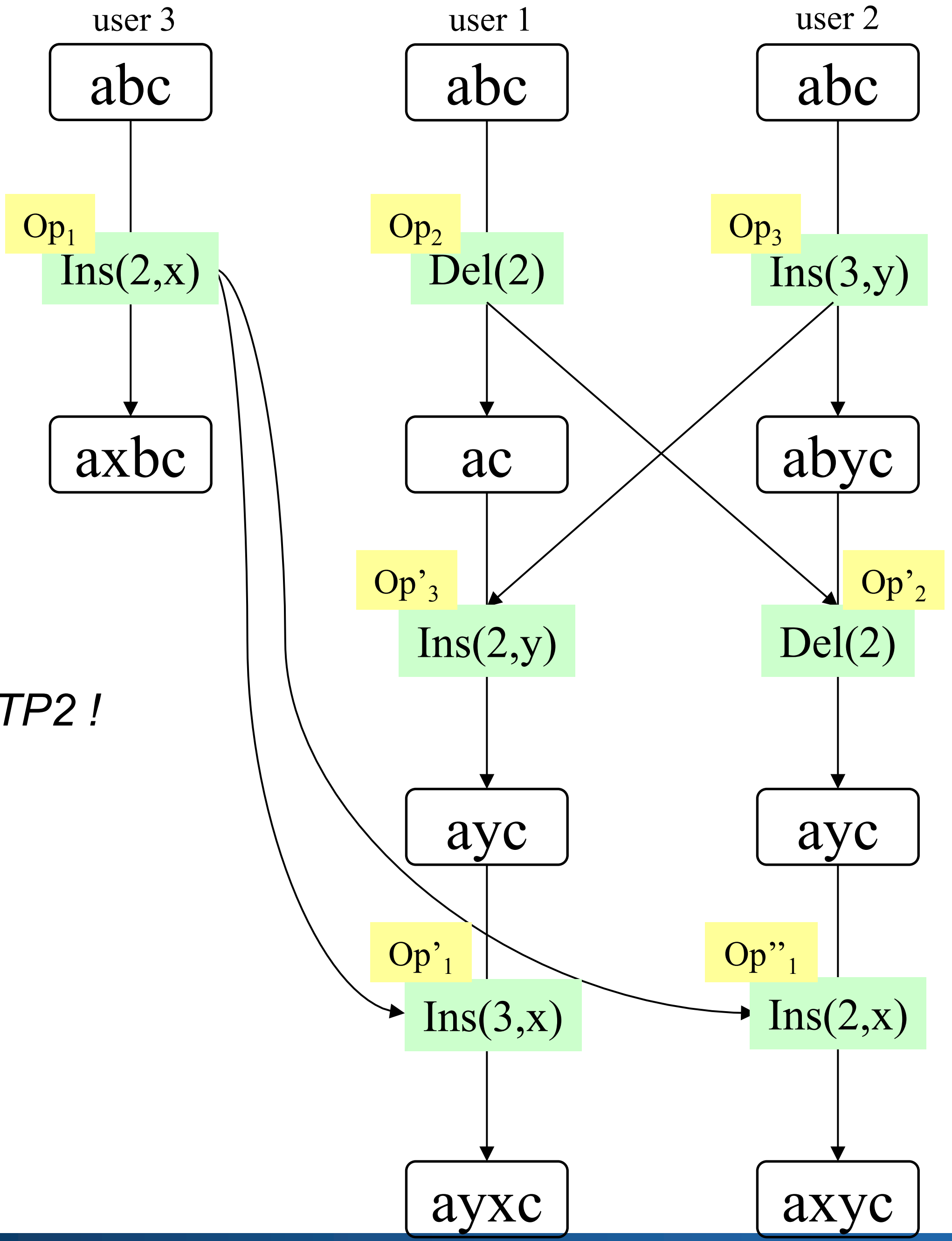


TP1 not respected !

Ressel transformation functions (*)

```
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

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()
```

(*) Ressel, M., Nitsche-Ruhland, D. & Gunzenhauser, R. (1996), An integrating, transformation oriented approach to concurrency control and undo in group editors, Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'96), Boston, Massachusetts, USA, pp. 288–297.



TP1 ok, but not TP2 !

Suleiman transformation functions (*)

Ins(p,c,a,b)

b – operations that have concurrently deleted a character before character c

a – operations that have concurrently deleted a character after character c

Two concurrent ins(p,c1,a1,b1) and ins(p,c2,a2,b2)

If $b1 \cap a2 \neq \emptyset$*, at generation* $p2 < p1$

If $a1 \cap b2 \neq \emptyset$*, at generation* $p1 < p2$

If $b1 \cap a2 = a1 \cap b2 = \emptyset$*, at generation* $p1 = p2$

(*) M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : (GROUP'97), pages 435.445, Phoenix, Arizona, United States, November 1997.

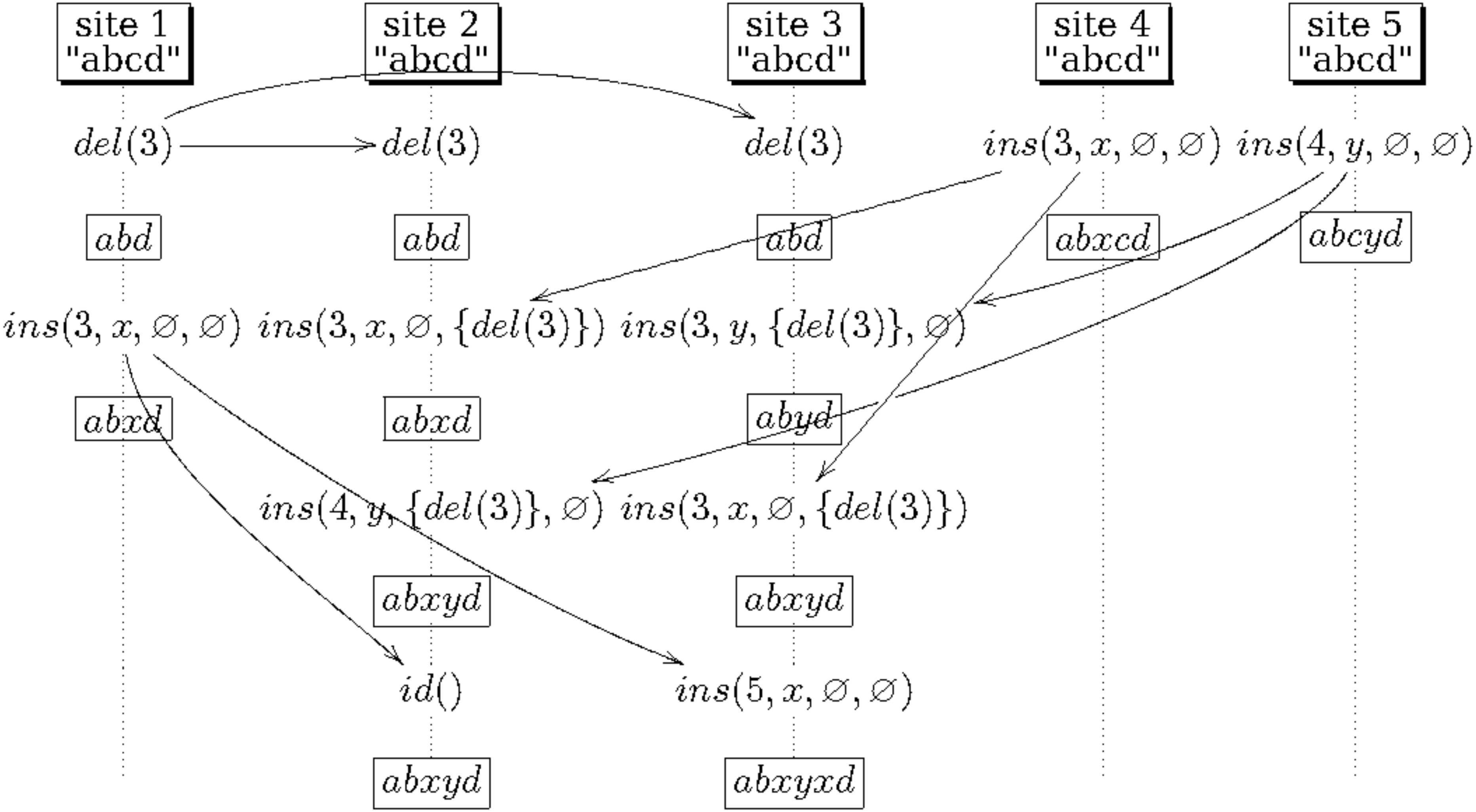
Suleiman transformation functions

```
T(Ins(p1,c1,a1,b1), Ins(p2,c2,a2,b2)) :-
    if ($p1 > p2$) then return Ins(p1+1,c1,a1,b1);
    else if ($p1 < p2$) then return Ins(p1,c1,a1,b1);
        else if ($p1 = p2$) then
            if ($b1 \cap a2 \neq \emptyset$) then return Ins(p1+1,c1,a1,b1);
            else if ($a1 \cap b2 \neq \emptyset$) then return Ins(p1,c1,a1,b1);
            else if (code(c1)>code(c2)) then return Ins(p1,c1,a1,b1);
            else if (code(c1)<code(c2)) then return Ins(p1+1,c1,a1,b1);
            else return id(Ins(p1,c1,a1,b1));
```

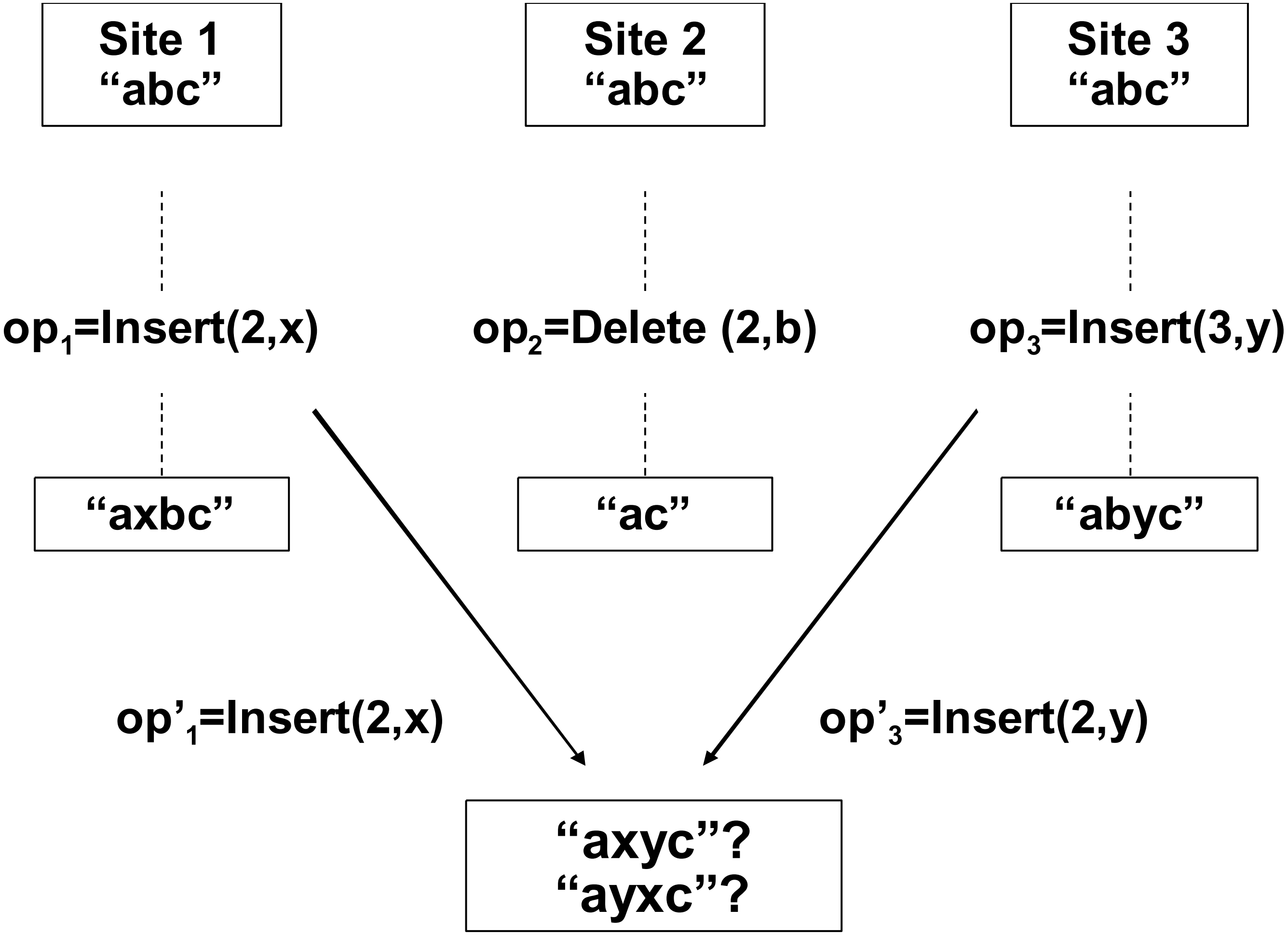
Suleiman transformation functions

```
T(Ins(p1,c1,a1,b1), Del(p2)) :-
    if (p1>p2) return Ins(p1-1,c1,b1+Del(p2),a1)
    else return Ins(p1,c1,b1,a1+Del(p2))
    endif

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

T(Del(p1), Ins(p2,c2,a2,b2)) :-
    if (p1<p2) return Del(p1)
    else return Del(p1+1)
```

Suleiman transformation functions



False-tie problem



Site 1
“abc”

Site 2
“abc”

Site 3
“abc”

op_1=Insert(2,x)

op_2=Delete (2,b)

op_3=Insert(3,y)

“axbc”

“ac”

“abyc”

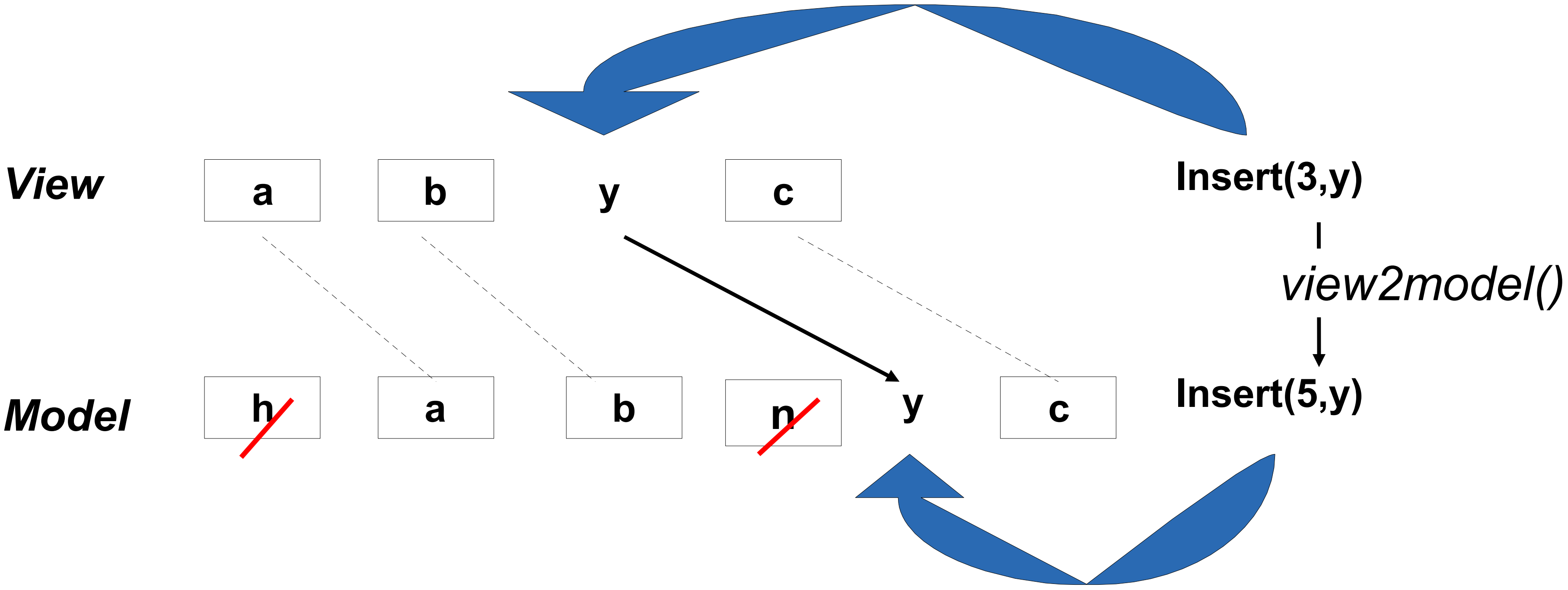
op'_1=Insert(2,x)

op'_3=Insert(2,y)

“axyc”?
“ayxc”?

TTF (Tombstone Transformation Functions) Approach (*)

- Keep “tombstones” of deleted elements



View a b y c

Insert(3,y)

view2model()

Model ~~h~~ a b ~~n~~ y c

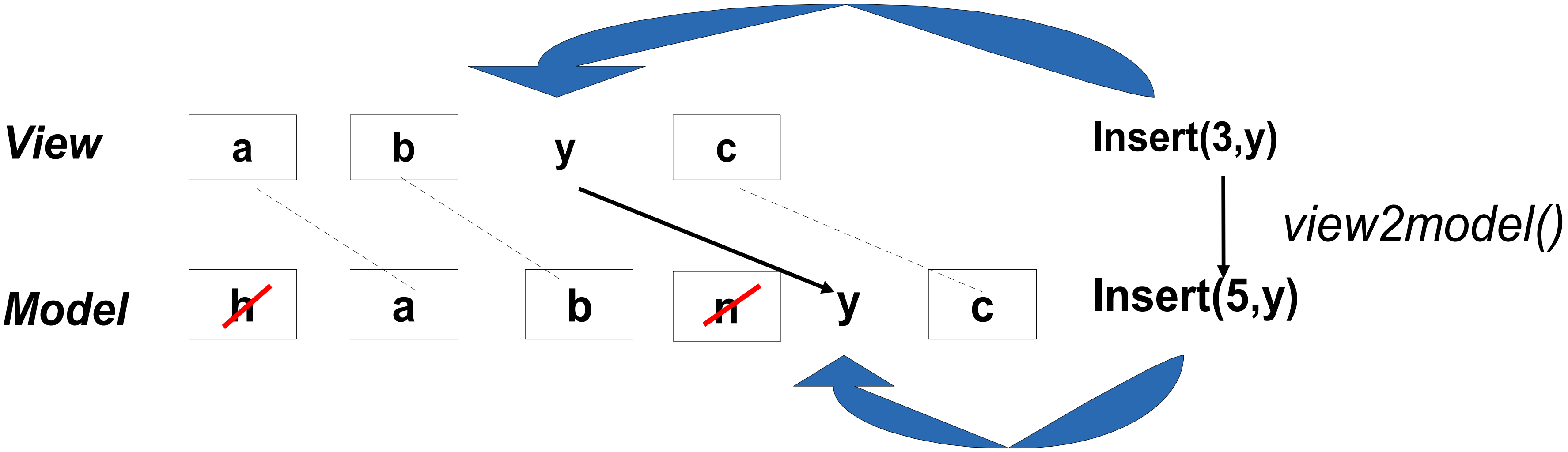
Insert(5,y)

(*) G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In The Second International Conference on Collaborative Computing : Networking,Applications and Worksharing (CollaborateCom 2006), Atlanta, Georgia,USA, November 2006. IEEE Press.

Tombstone Transformation Functions

```
T(Insert(p1,el1,sid1), Insert(p2,el2,sid2)){
    if(p1<p2) return Insert(p1,el1,sid1)
    else if(p1=p2 and sid1<sid2) return Insert(p1,el1,sid1)
    else return Insert(p1+1,el1,sid1)
}
T(Insert(p1,el1,sid1), Delete(p2,el2,sid2)){
    return Insert(p1,el1,sid1)
}
T(Delete(p1,sid1), Insert(p2,sid2)){
    if(p1<p2) return Delete(p1,sid1)
    else return Delete(p1+1,sid1)
}
T(Delete(p1,sid1), Delete(p2,sid2)){
    return Delete(p1,sid1)
}
```

Compacted storage model



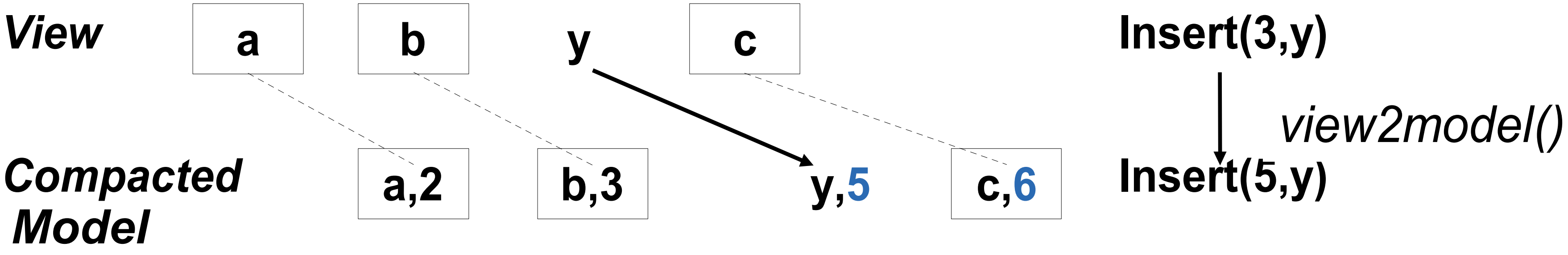
View a b y c

Model ~~h~~ a b ~~n~~ y c

Insert(3,y)

view2model()

Insert(5,y)



View a b y c

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

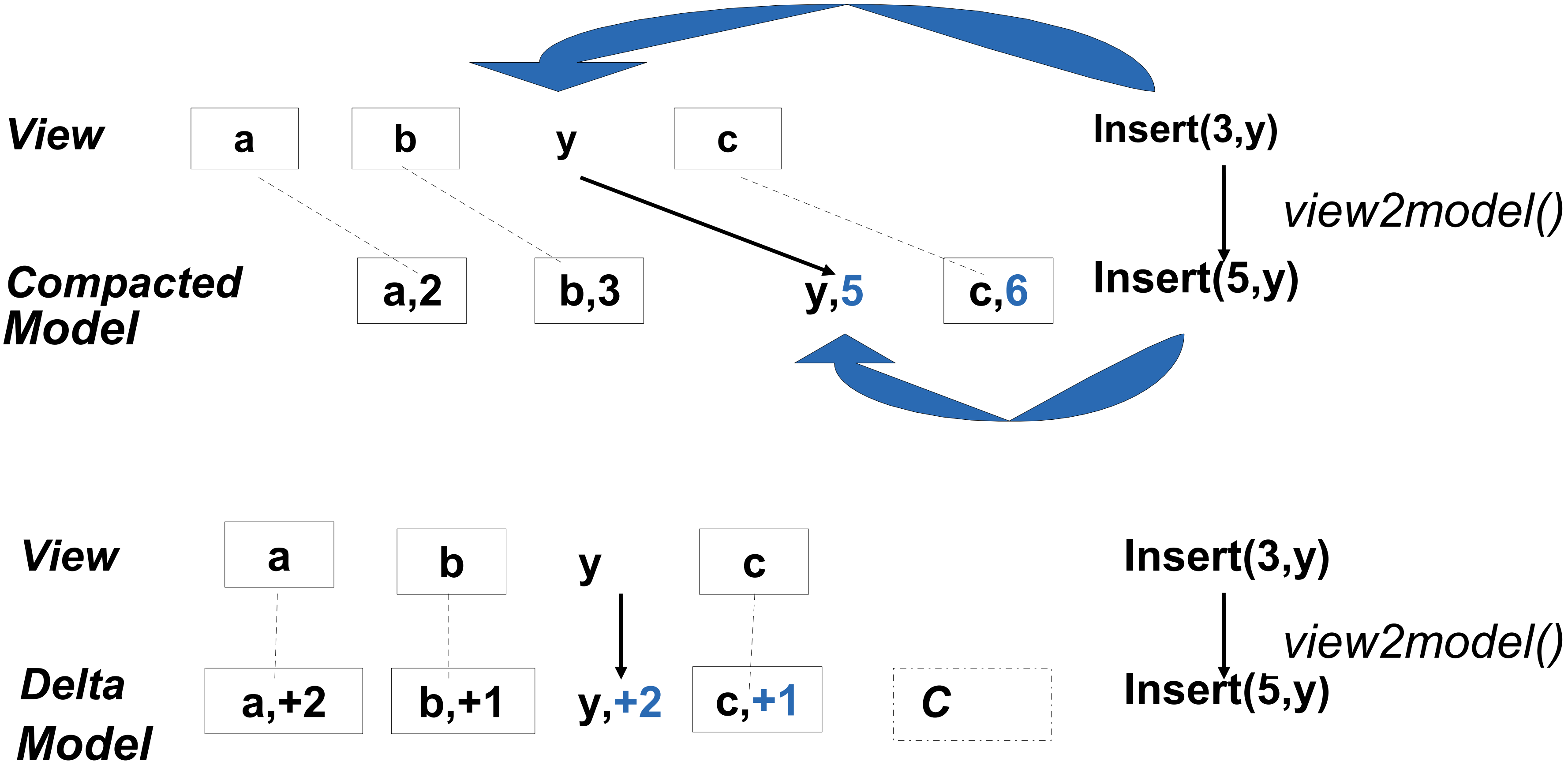
Insert(3,y)

view2model()

Insert(5,y)

- Compacted model = sequence of (character, abs_pos)

Delta storage model



View a b y c

Insert(3,y)

view2model()

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

Insert(5,y)

View a b y c

Insert(3,y)

view2model()

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

Insert(5,y)

- Delta model = sequence of (character, offset)

Models 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
- Our observations
  - View2model can be optimised (caret position)
  - Overhead of view2model is not significant