Distributed Systems
Operational Transformation

Dr.-Ing. Matthäus Wander

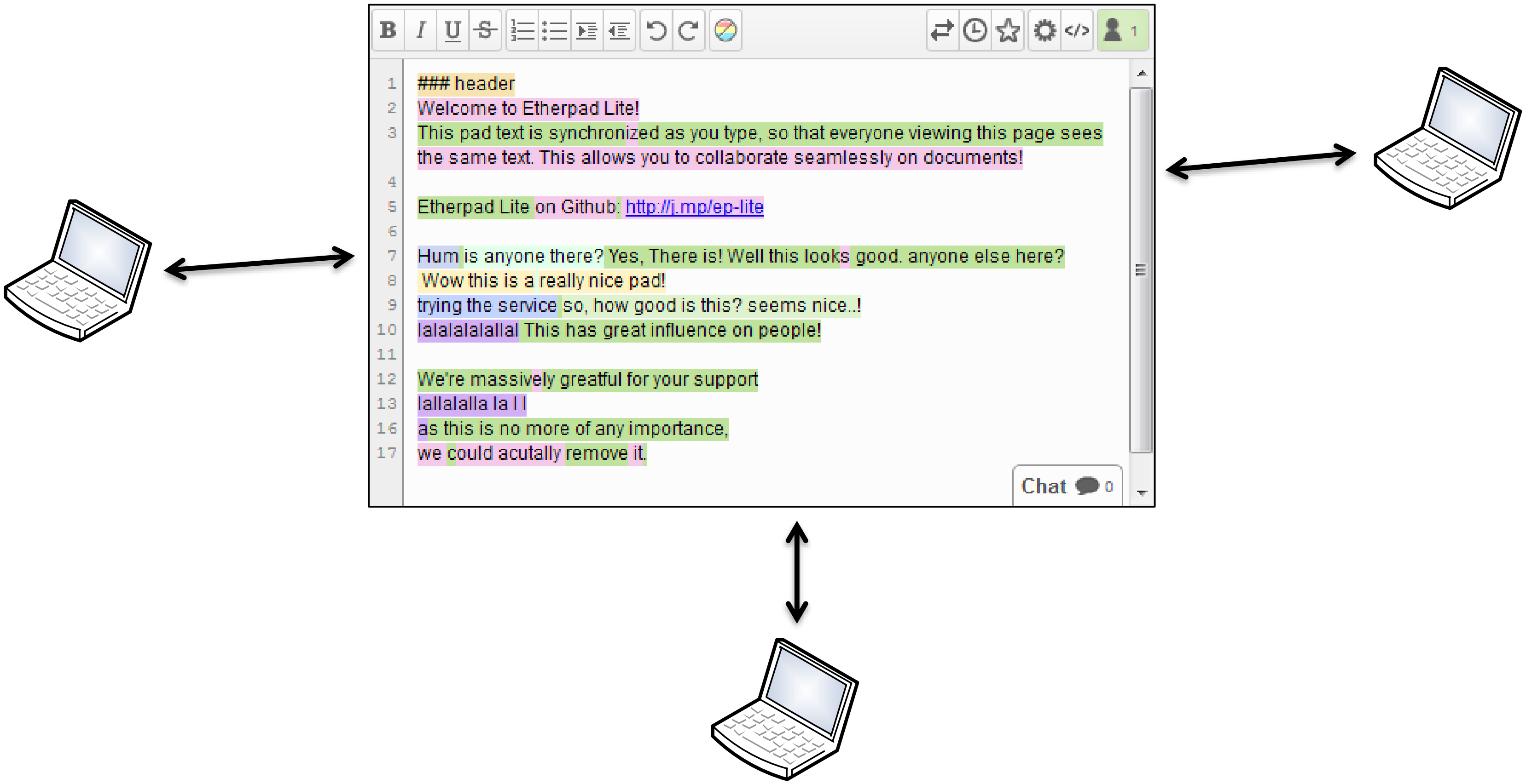
University Duisburg-Essen

Concurrency Control

- Locking
  - Lock object before accessing it
  - Conflicting operations will wait
- Transactions
  - Lock access to multiple objects in the right order
- Optimistic concurrency control
  - Don‘t lock, but abort&retry transaction on conflict
- Problem solved, right?
  - What if conflicts are common in our application?

Example: Etherpad

- Multiple users editing text at the same time

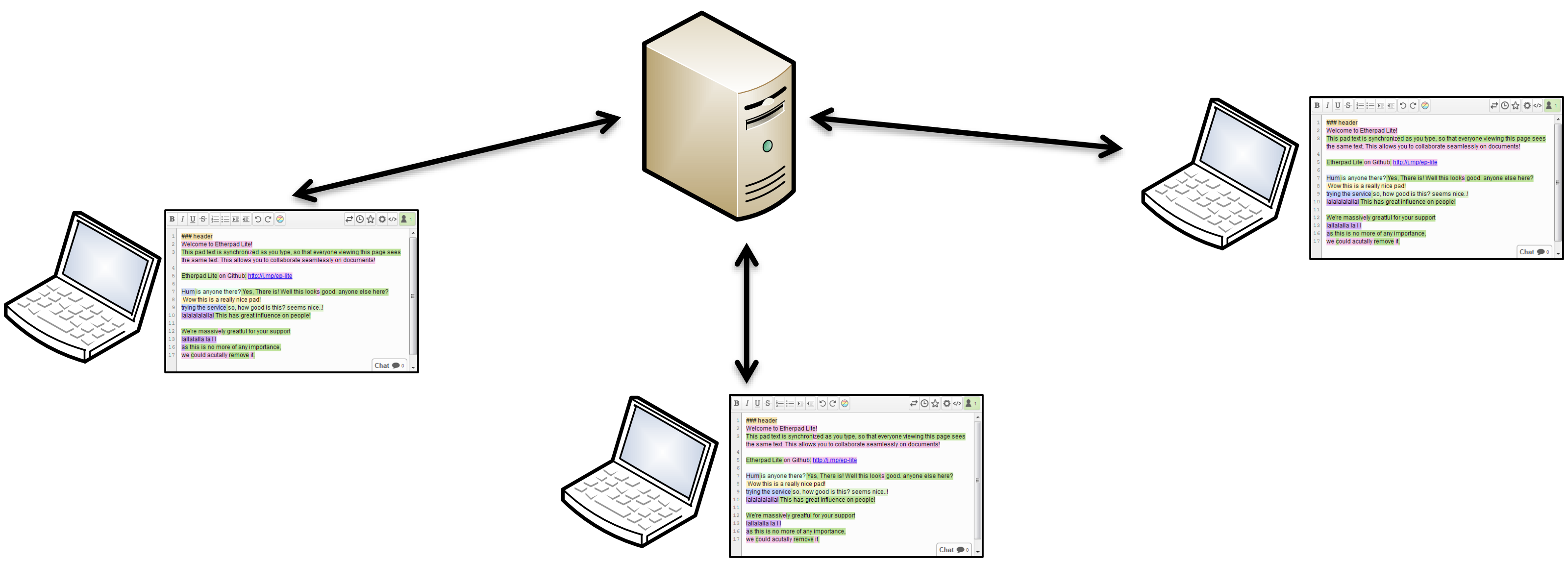


Groupware

- Groupware or collaborative software
  - Multiple users working in a session on the same data
- Properties:
  - Distributed system with replicated data
  - Same user interface
  - Eventually consistent view on the same data
  - Highly-interactive (response user interface)
  - Real-time (user actions quickly update others‘ views)
  - Collaboration (users are working together)

System Model

- Each user has view on her copy of data
  - Changes made locally quickly change the local view
  - Changes are distributed peer-to-peer or server-based
  - Document changes are distributed as operations



Comparison with other Concurrency Control

- Why not locking or transactions?
    - Network delay when waiting for lock
    - Waiting time until data unlocks
    - Slow, unresponsive user interface ☹
    - Breaks high interactivity
- Why not optimistic concurrency control?
    - Conflicts typical during collaborative editing
    - Transaction abort ⇨ user action reverted
    - Frustrating collaboration ☹

System Model (2)

- Groupware system G = <S, O>
  - S: set of sites
    i.e. application instances running on user machines
  - O: set of parametrized operators
    i.e. possible operations on data
- Each site consists of:
  - Application process
  - Site object, i.e. copy the shared data
  - Unique site identifier

System Model (3)

- Example: sites $\{S_1, S_2, S_3\}$ edit a text string
  - Two operators $\{O_1, O_2\}$
  - O_1 := insert[X; P] insert character X at position P
  - O_2 := delete[P] delete character at position P
- We apply instances of operations on our data
  - Say we have $o := O_1[x; 3]$
  - Assume position index starts at 1
  - o("abc") gives us "abxc"

Site Activities

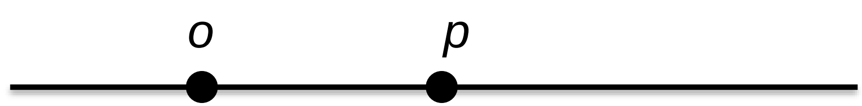
- Operation generation
  - User actions generate operation requests, which are broadcasted to other sites
- Operation reception
  - Sites listen and receive operation requests from other sites
- Operation execution
  - Sites execute operation requests on their site object

Assumptions

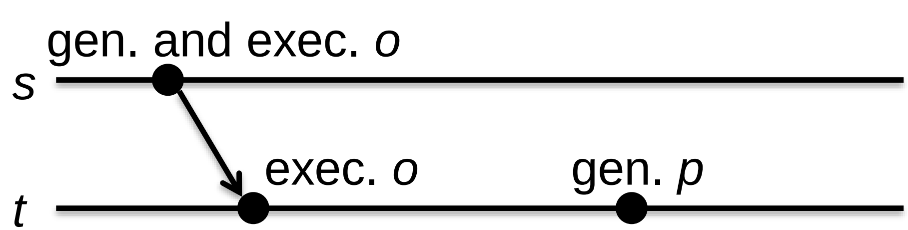
1. The number of sites is constant
   - Users can‘t join/leave while the algorithm is running
   - A usable system will have to relax this assumption
2. Messages are received exactly once and without error
   - Algorithm does not assume FIFO ordering
3. It is not possible for a message to be executed before it is generated

Precedence

- Given two operations *o*, *p* at one site:
  - *o* → *p*, iff *o* was generated before *p*



- Given two operations *o*, *p* at two sites *s*, *t* :
  - *o* → *p*, iff *o* was generated at *s* and executed at *t* before *p* was generated at *t*



Properties and Correctness

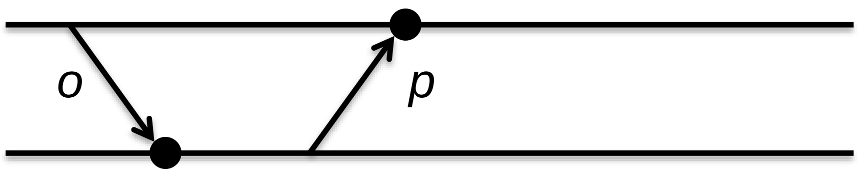
- **Precedence** Property:
  - For all *o*, *p* with *o* → *p*, all sites execute o before p
- Definition: groupware session is **quiescent** if all generated operations have been executed
  - i.e. no pending requests; system waiting for input
- **Convergence** Property:
  - When quiescent, data objects are identical at all sites
- Groupware system is correct iff precedence and convergence properties are satisfied

Precedence vs. Responsiveness

- As long as we adhere to the partial ordering of precedence, our system will be correct
- Use logical clocks or snapshot algorithm?
  - Agree on a total order by exchanging timestamps
  - Some coordination between sites required
  - Introduces delay ⇨ unresponse application ☹
- We need to execute operations as quickly as possible

Problem: Overlapping Operations

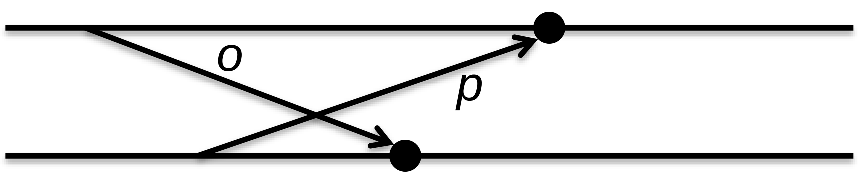
- Can we execute operations instantly on generation and reception?
  - e.g. *o*:=delete[3], *p*:=delete[2] on "abcd"



p(o("abcd") = p("abd") = "ad"

p(o("abcd") = p("abd") = "ad"

  - Sometimes yes, but in general no
  - Overlapping, non-commutative operations:

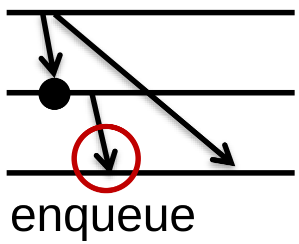


p(o("abcd") = p("abd") = "ad"

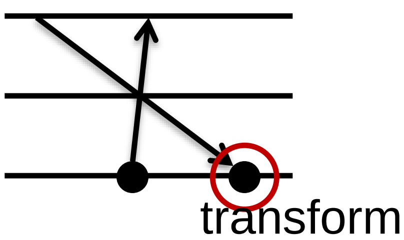
o(p("abcd") = o("acd") = "ac"

Operational Transformation Algorithm

- Upon operation generation (due to user action):
  - Execute the operation locally
  - Send operation to all other sites
- Upon operation reception:
  - Has the sender executed another operation that the receiver has not? Future op. ⇨ enqueue and wait



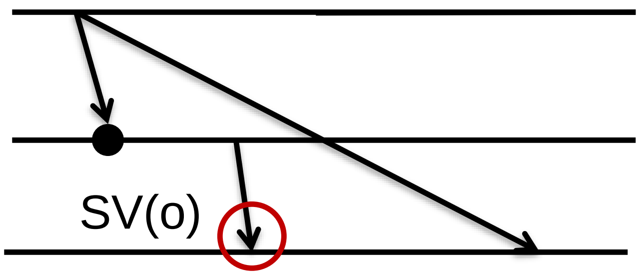
  - Has the receiver executed another operation that the sender has not? Past op. ⇨ transform and execute



  - Otherwise ⇨ execute

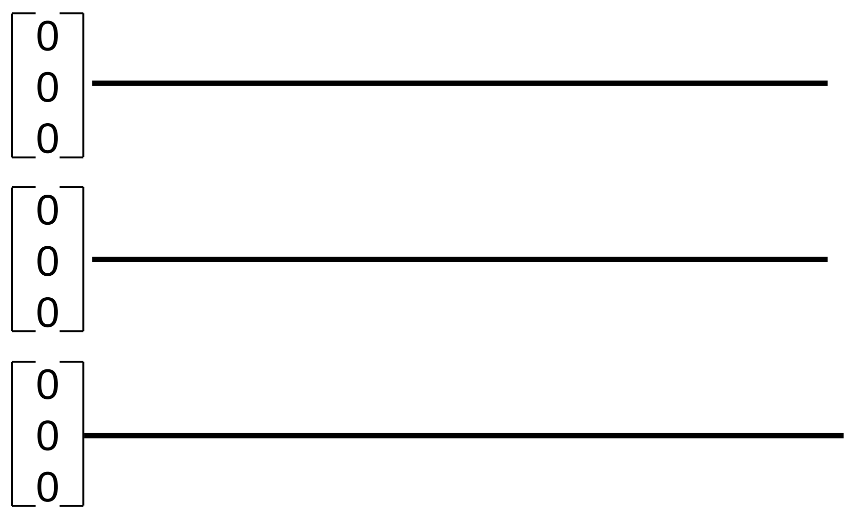
Operational Transformation Algorithm (2)

- How long do we have to wait?
  - How do we know which future operations we need?
- Use **state vector** to find out
  - A type of vector clocks, which give us the whole history of events that happened before an event e
  - Send state vector together with operation *o*, which indicates which operations happened before *o*



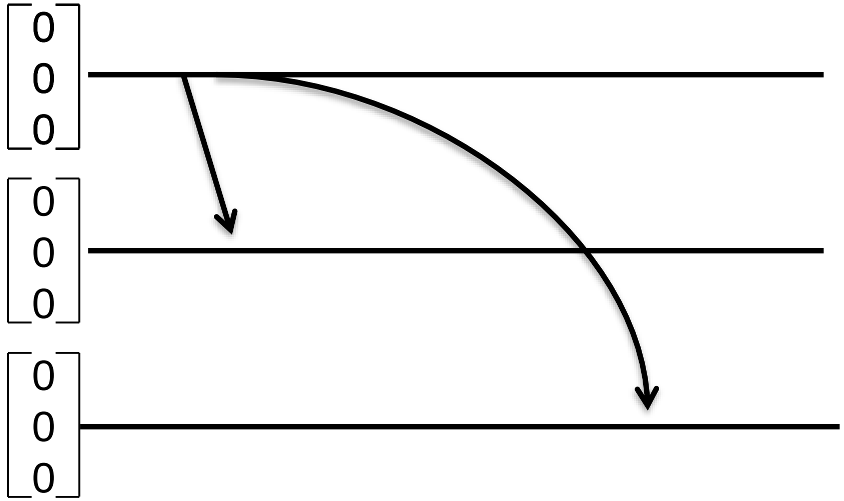
Operational Transformation Algorithm (3)

- SV(s): state vector for site s
- SV(s)[*i*]: *i*–th component indicates number of operations from *i* that were executed at site *s*
  - *Beware: slightly different semantics than vector clocks*
  - Increment only upon execution, not send/receive
  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t* :
    - SV(s)[t] := SV(s)[t] + 1



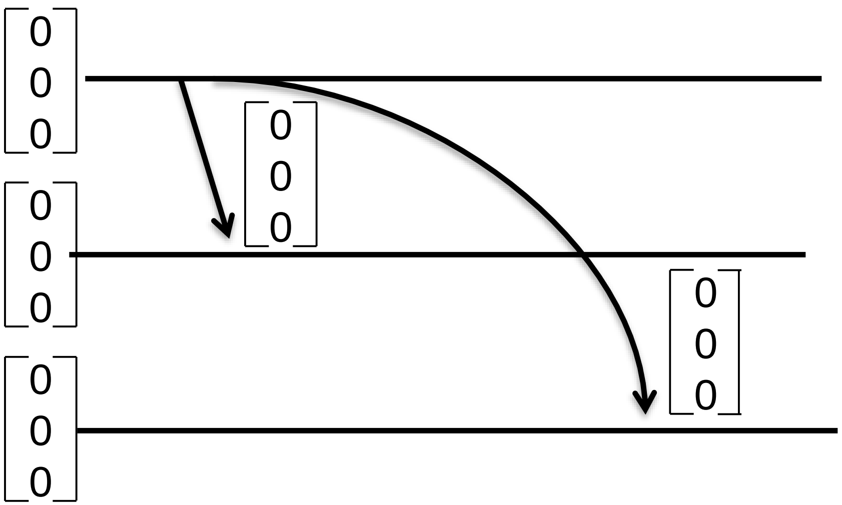
Operational Transformation Algorithm (3)

- SV(s): state vector for site s
- SV(s)[*i*]: *i*–th component indicates number of operations from *i* that were executed at site *s*
  - *Beware: slightly different semantics than vector clocks*
  - Increment only upon execution, not send/receive
  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t* :
    - SV(s)[t] := SV(s)[t] + 1



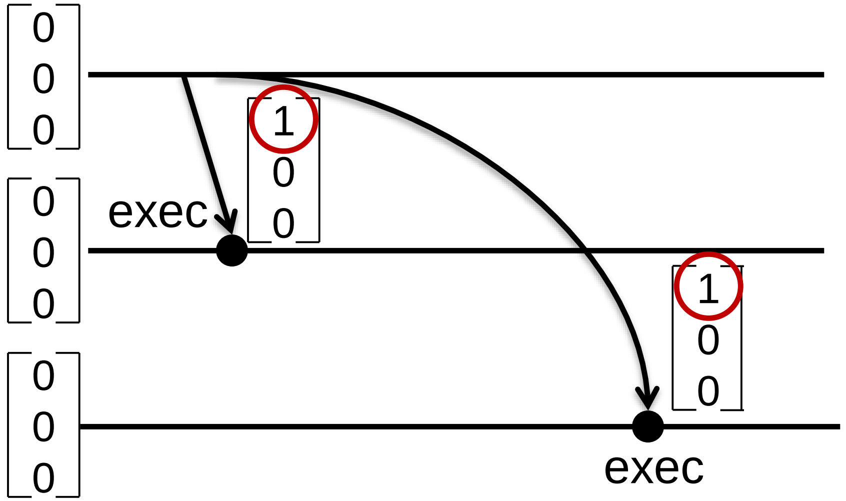
Operational Transformation Algorithm (3)

- SV(s): state vector for site s
- SV(s)[*i*]: *i*–th component indicates number of operations from *i* that were executed at site *s*
  - *Beware: slightly different semantics than vector clocks*
  - Increment only upon execution, not send/receive
  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t*:
    - SV(s)[t] := SV(s)[t] + 1



Operational Transformation Algorithm (3)

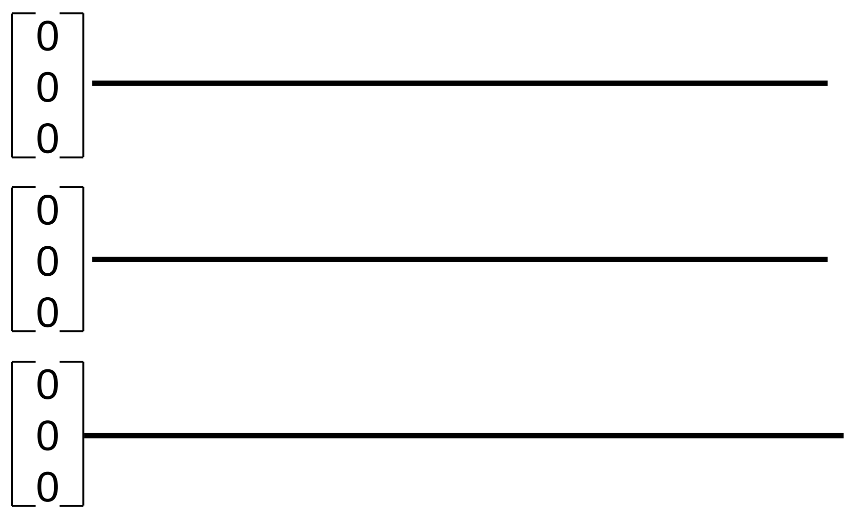
- SV(s): state vector for site s
- SV(s)[*i*]: *i*–th component indicates number of operations from *i* that were executed at site *s*
  - *Beware: slightly different semantics than vector clocks*
  - Increment only upon execution, not send/receive
  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t* :
    - SV(s)[t] := SV(s)[t] + 1



Operational Transformation Algorithm (4)

- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute

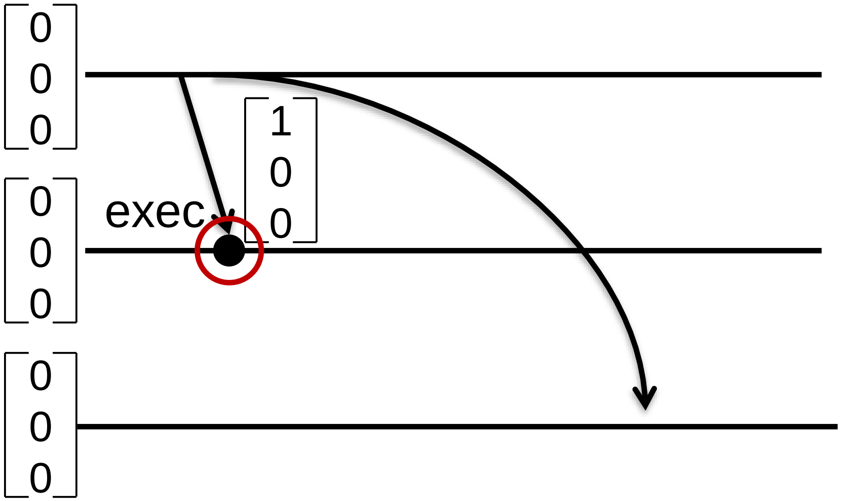
  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t* :
    - SV(s)[t] := SV(s)[t] + 1



Operational Transformation Algorithm (4)

- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute

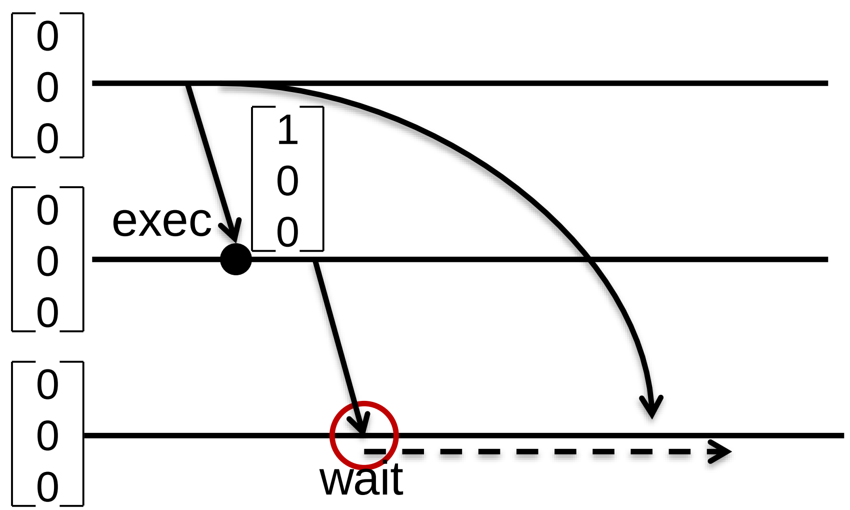
  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t* :
    - SV(s)[t] := SV(s)[t] + 1



Operational Transformation Algorithm (4)

- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute

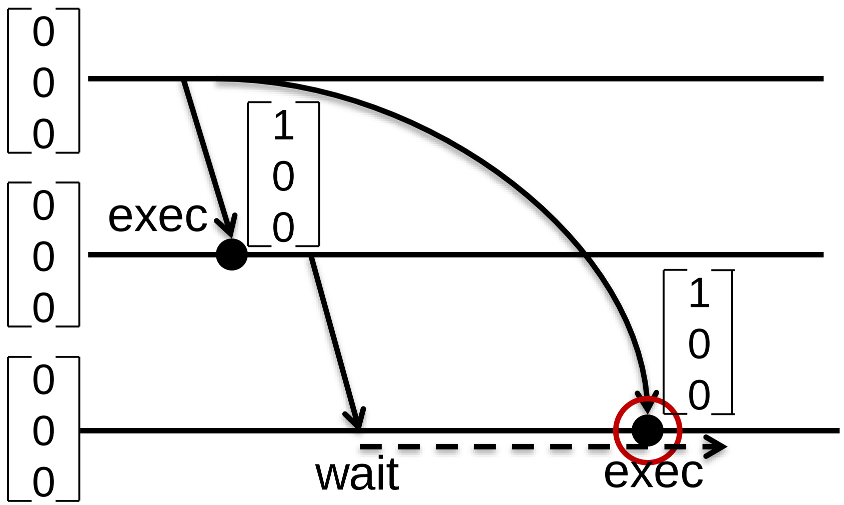
  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t* :
    - SV(s)[t] := SV(s)[t] + 1



Operational Transformation Algorithm (4)

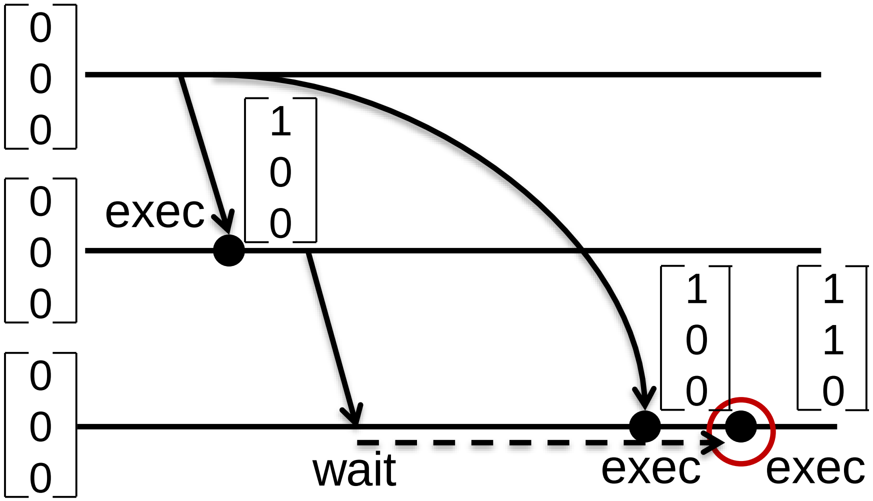
- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute

  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t* :
    - SV(s)[t] := SV(s)[t] + 1



Operational Transformation Algorithm (4)

- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute

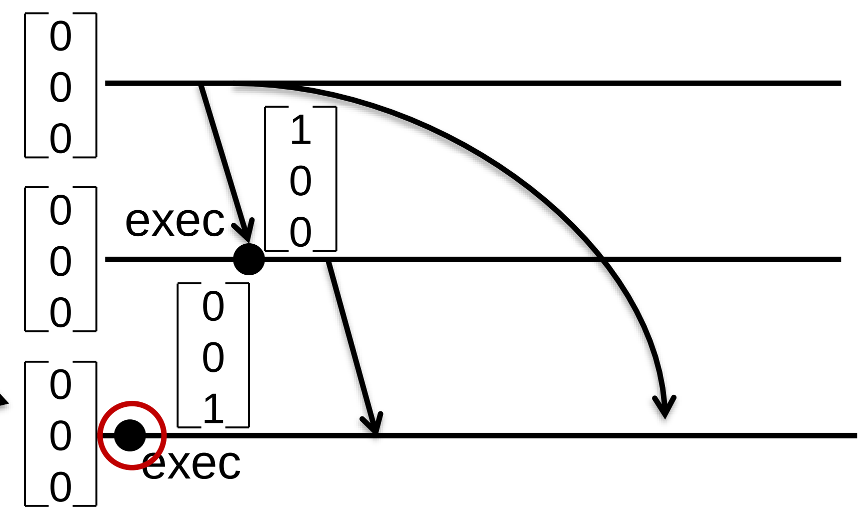


  - Generate: send SV(s)
  - Receive: no change
  - *s* executes *o* from *t*:
    - SV(s)[t] := SV(s)[t] + 1

Operational Transformation Algorithm (5)

- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute

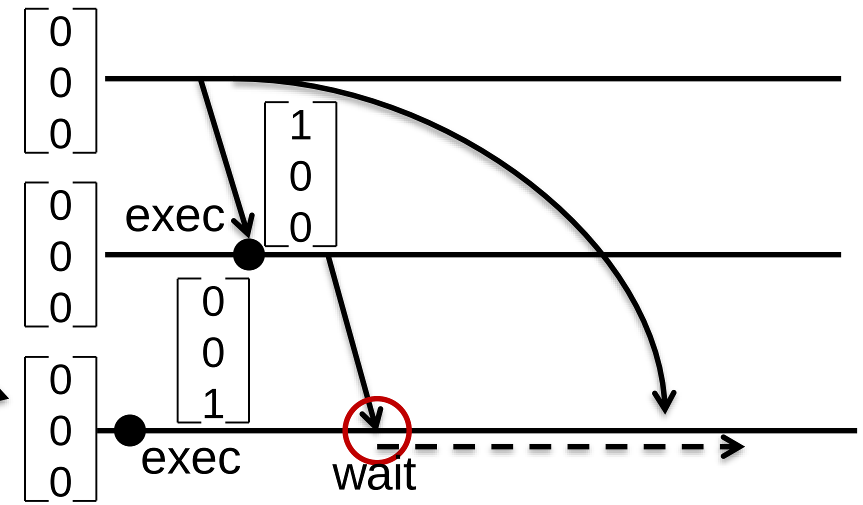
  - Another example
  - Assume an operation processed here



Operational Transformation Algorithm (5)

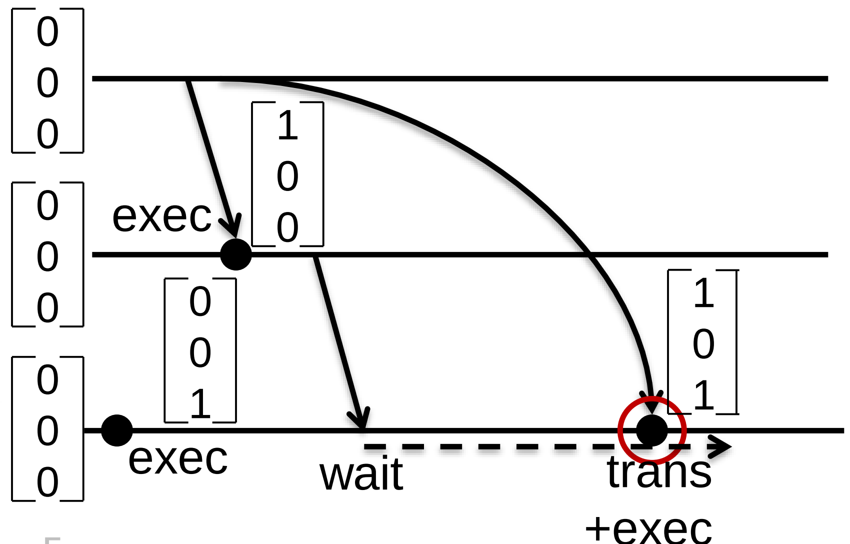
- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute

  - Another example
  - Assume an operation processed here



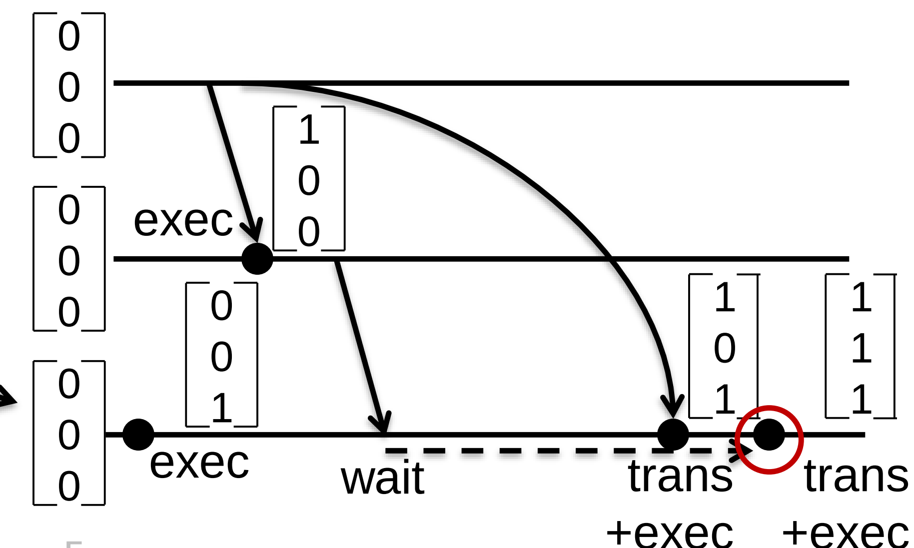
Operational Transformation Algorithm (5)

- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute
  - Another example
  - Assume an operation processed here



Operational Transformation Algorithm (5)

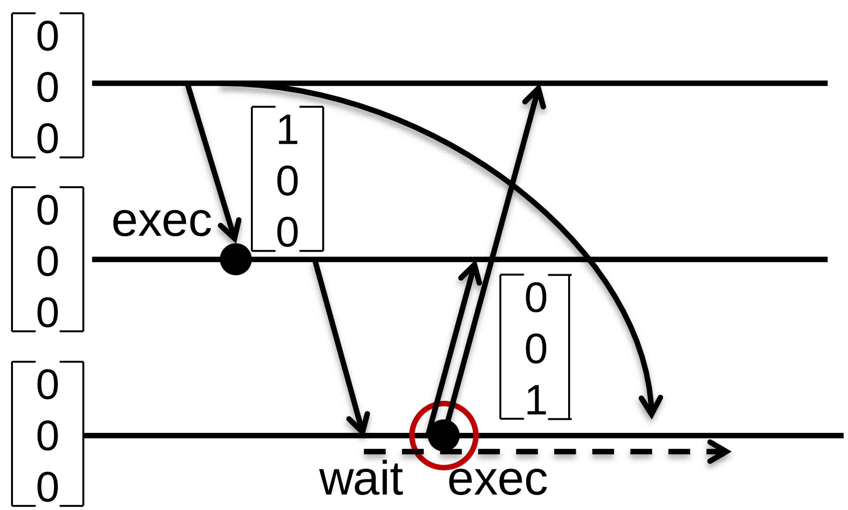
- When to execute? Site s receives o from t:
  - If SV(s) < SV(o) or SV(s) || SV(o): enqueue and wait
  - If SV(s) = SV(o): execute
  - If SV(s) > SV(o): transform and execute



  - Another example
  - Assume an operation processed here

Operational Transformation Algorithm (6)

- Site *s* generates operation *p*
  - Send SV(p):=SV(s) to all other sites
- Then executes *p* locally
  - Which updates SV(s) **after** sending *p*
  - SV(s)[s] := SV(s)[s] + 1
- Can execute local operation *p* always, even if received operations are queued
  - Because SV(s) = SV(p)



Transformation Matrix

- Two operations *o*, *p* are not commutative
  - i.e. different order yields different data
- Transform *o*, *p* into new operation *o'* or *p'*
  - If two sites execute *o* and *p* concurrently, they then execute a transformed *o'*, *p'* to get the same result
  - p' := T(p, o)   o' := T(o, p)
  - Site s executes *o*, transforms *p*, executes *p'*
  - Site t executes *p*, transforms *o*, executes *o'*
  - It holds: p'(o(data)) = o'(p(data))

Transformation Matrix (2)

- We need a transformation for any two operators
- Example: two operators $\{O_1, O_2\}$
  - O_1 := insert[X; P] insert character X at position P
  - O_2 := delete[P] delete character at position P

T	O_1: insert	O_2: delete
O_1: insert	T_{11}	T_{12}
O_2: delete	T_{21}	T_{22}

- The matrix is application-specific and grows with each new operator
  - Quite complex implementation

Example: Transform Insert/Insert

- o := insert[X_o; P_o] insert X_o at position P_o

```
o‘ := T11(o, p):
  if Po < Pp:
    // insert char left of p: no change
    o‘ := insert[Xo; Po]
  else if Po > Pp:
    // insert char right of p: position + 1
    o‘ := insert[Xo; Po + 1]
  else:
    // identical operations cancel each other out
    if Xo = Xp:
      o‘ := identity() // do nothing
    else:
      … // use some tie-breaking mechanism
```

Example: Transform Insert/Insert (2)

- o := insert[A; 1],   p := insert[B; 2]
  - o‘ := T_{11}(o, p) = insert[A; 1]
  - p‘ := T_{11}(p, o) = insert[B; 3]
  - o‘(p("xyz")) = o‘("xByz") = "AxByz"
  - p‘(o("xyz")) = p‘("Axyz") = "AxByz"
- o := insert[W; 1],   p := insert[W; 1]
  - o‘ := T_{11}(o, p) = identity()
  - p‘ := T_{11}(p, o) = identity()
  - o‘(p("xyz")) = o‘("Wxyz") = "Wxyz"
  - p‘(o("xyz")) = p‘("Wxyz") = "Wxyz"

Conclusions

- Operational Transformation is optimistic concurrency control without aborts
  - Conflicting operations are transformed
  - Suitable for highly-interactive applications like groupware
- Algorithm is generic, but transformation matrix is application-specific
  - Algorithm fails to converge in certain scenarios
  - Problem („TP2 convergence“) solved by later algorithms