Agenda

- Optimistic replication
  - Operational transformation: algorithms
  - Conflict-free Replicated Data Types
SOCT2 algorithm(*)
General control algorithm

a) The initial history buffer

Equivalent HB

operations preceding the remote operation

operations concurrent with the remote operation

b) Principle of integration

GOT algorithm(*)

• Does not need to satisfy TP1 and TP2
• Requires a global serialisation order
  ▪ Sum of state vector components
  ▪ If equality, then priority on sites
• Requires undo/redo mechanism
• Undo/redo very costly

SOCT4 algorithm(*)

- Does not use undo/redo mechanism
- Eliminates TP2, but requires TP1
- Does not need state vectors
- 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 all preceding operations were executed
- Transformations performed by each site

SOCT4 algorithm

Histoire $H(n)$ avant l'intégration $o_1$ $o_2$ ... $o_i$

Intégration de $o_{i+1}$ $o_1$ $o_2$ ... $o_i$

transpositions en avant des opérations locales

History $H(n)$ après intégration $o_1$ $o_2$ ... $o_i$ $o_{i+1}$ $o'_1$ $o'_2$ ... $o'_{i+1}$

opérations locales transposées en avant

transposition en avant de $o_{i+1}$

opérations locales en attente de diffusion

seq

op$_{i+1}$ $o_{L_1}$ $o_{L_m}$
So6- variation of SOCT4 algorithm(*)

```
Sync (log , N_s ) :-
  while ( ( op_r = getOp ( N_s +1 ) ) != 0 ) 2
    for ( i = 0 ; i < log.size(); i ++ ) 4
      op_l = log [ i ]; 4
      log [ i ] = T ( op_l , op_r ) 6
      op_r = T ( op_r , op_l ) 6
    endfor 8
    execute op_r 8
    N_s = N_s + 1 10
  endwhile 10
  for ( i = 0 ; i < log.size(); i ++ ) 12
    op_l = log [ i ]; 14
    if send ( op_l , N_s + 1 ) then 14
      N_s = N_s + 1 16
    else 16
      error ' need to synchronize 16
    endif 16
    endif 18
```

getOp (ticket) retrieves operation identified by timestamp ticket

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

**So6 algorithm**

<table>
<thead>
<tr>
<th>Site1, Ns=0</th>
<th>Site2, Ns=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1</td>
<td>op3</td>
</tr>
<tr>
<td>op2</td>
<td>op4</td>
</tr>
<tr>
<td>s1=synchronize</td>
<td>s2=synchronize</td>
</tr>
<tr>
<td>s3=synchronize</td>
<td></td>
</tr>
</tbody>
</table>

- **At s1**
  - `sync([op1,op2],0)`
  - Send op1, op2
  - Ns=2

- **At s2**
  - `sync([op3,op4],0)`
  - `op'1=T(op1,op3)`
  - `op'3=T(op3,op1)`
  - `op'1=T(op'1,op4)`
  - `op'4=T(op4,op'1)`
  - `op'2=T(op2,op'3)`
  - `op'3=T(op'3,op2)`
  - `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 s3**
  - `sync([],2)`
  - op’’3 and op’’4 are executed
  - Ns=4
### SO6 algorithm

<table>
<thead>
<tr>
<th>Site 1, Ns=0</th>
<th>Site 2, Ns=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1</td>
<td>op3</td>
</tr>
<tr>
<td>op2</td>
<td>op4</td>
</tr>
<tr>
<td>s1=synchronize</td>
<td>s2=synchronize</td>
</tr>
<tr>
<td>s3=synchronize</td>
<td></td>
</tr>
</tbody>
</table>

**Site 1**

- op1
- op2
- $op''3 = T(T(op3,op1), op2)$
- $op''4 = T(T(op4, op'1), op'2)$

**Site 2**

- op3
- op4
- $op''1 = T(T(op1, op3), op4)$
- $op''2 = T(T(op2, op'3), op'4)$
SO6 algorithm

- create(obj)
- change_color(obj, color)
- del(obj)

- \( T(\text{create}(\text{obj}),_) :- \)
  return create(obj)
- \( T(\text{change}_\text{color}(\text{obj1},c), \text{del}(\text{obj2})) :- \)
  if (obj1==obj2) then return id()
  else return \( \text{change}_\text{color}(\text{obj1},c) \)
- \( T(\text{del}(\text{obj1}), \text{del}(\text{obj2})) :- \)
  if (obj1==obj2) then return id()
  else return \( \text{del}(\text{obj1}) \)
- \( T(\text{del}(\text{obj1}), \text{change}_\text{color}(\text{obj2},c)) :- \)
  return \( \text{del}(\text{obj1}) \)
- \( T(\text{change}_\text{color}(\text{obj1},c1), \text{change}(\text{obj2},c2)) :- \)
  if (obj1==obj2) then
    if (c1>c2) then return \( \text{change}_\text{color}(\text{obj1},c1) \)
    else return id()
  else return \( \text{change}_\text{color}(\text{obj1},c1) \)
- \( T(\text{id}(),_) :- \text{id}() \)
- \( T(\text{op},\text{id}()) :- \text{return op} \)
\[
\begin{align*}
\text{op}’1 &= T(\text{op1}, \text{op3}) = \text{del}(\text{obj1}) \\
\text{op}’3 &= T(\text{op3}, \text{op1}) = \text{id}() \\
\text{op}’’1 &= T(\text{op}’1, \text{op4}) = \text{del}(\text{obj1}) \\
\text{op}’4 &= T(\text{op4}, \text{op}’1) = \text{change\_color}(\text{obj2}, \text{blue}) \\
\text{op}’2 &= T(\text{op2}, \text{op}’3) = \text{change\_color}(\text{obj2}, \text{red}) \\
\text{op}’’3 &= T(\text{op}’3, \text{op2}) = \text{id}() \\
\text{op}’’2 &= T(\text{op}’2, \text{op}’4) = \text{change\_color}(\text{obj2}, \text{red}) \\
\text{op}’’4 &= T(\text{op}’4, \text{op}’2) = \text{id}()
\end{align*}
\]
Jupiter algorithm(*)

- Used in Google Drive
- Requires a central server
- Eliminates TP2, but requires TP1
- Does not need state vectors
- Transformations done on the server + client side

Jupiter algorithm

- \( \text{xform}(c, s) = \{c’, s’\} \)
- \( \text{xform}(\text{del } x, \text{del } y) = \)
  - \( \{\text{del } x-1, \text{del } y\} \) if \( x>y \)
  - \( \{\text{del } x, \text{del } y-1\} \) if \( x<y \)
  - \( \{\text{no-op, no-op}\} \) if \( x=y \)
Jupiter algorithm

(a)

(b)
Jupiter algorithm
Jupiter algorithm
2 sites

int myMsgs = 0; /* number of messages generated */
int otherMsgs = 0; /* number of messages received */
queue outgoing = {};

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 message and the ones in the queue. */
        {msg, outgoing[i]} = xform(msg, outgoing[i]);
    }
    apply msg.op locally;
    otherMsgs = otherMsgs + 1;
}
\[
\begin{align*}
\text{Site 1} & \quad \text{Site 2} \\
\text{Exec. op1} & \quad \text{Exec. op3} \\
\text{Exec. op2} & \quad \text{Exec. op' 1} \\
\text{Exec. op' 3} & \quad \text{Exec. op4} \\
\text{Exec. op' 4} & \quad \text{Exec. op' 2} \\
\end{align*}
\]
**Jupiter algorithm – generalisation n**

**Clients**

```
apply msg.op locally;

for (c in client list) {
    if (c != client) {
        send(c, msg);
    }
}
```
Jupiter algorithm

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

Site 1
“abc”

Site 2
“abc”

Site 3
“abc”

\[\text{op}_1 = \text{Insert}(2, x)\]

“axbc”

\[\text{op}_2 = \text{Delete}(2, b)\]

“ac”

\[\text{op}_3 = \text{Insert}(3, y)\]

“abye”

\[\text{op}'_1 = \text{Insert}(2, x)\]

“axyc”? “ayxc”?  

\[\text{op}'_3 = \text{Insert}(2, y)\]
Conflict-free Replicated Data Types (CRDT) (*)

- Design operations to be commutative by construction

- Abstract data types
  - Designed to be replicated at multiple sites
  - Any replica can be modified without coordination
  - State convergence is guaranteed

- 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

\[
\begin{array}{c}
\text{s}_1 \rightarrow \text{s}_1.\text{u(\alpha)} \\
\text{s}_2 \rightarrow \text{s}_2.\text{u(\beta)} \\
\text{s}_2 \rightarrow \text{s}_2.\text{m(s}_1) \\
\text{s}_1 \rightarrow \text{s}_1.\text{m(s}_2) \\
\text{s}_2 \rightarrow \text{s}_2.\text{m(s}_2) \\
\text{s}_3 \rightarrow \text{s}_3.\text{m(s}_2)
\end{array}
\]
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
    $\forall m', x \leq m' \land y \leq m' \Rightarrow x \leq m \land y \leq m \land m \leq m'$

• 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: $\max(\max(x, y), z) = \max(x, \max(y, z))$
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 the 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., \( s \cdot m(s') = s \sqcup s' \) (delivery order is not important)
  3. State is monotonically non-decreasing across updates, i.e., \( s \leq s \cdot u \) (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 \((e, \alpha)\)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 \(s \cdot u'\) (resp. \(s \cdot u\))
  - \(s \cdot u \cdot u' \equiv s \cdot u' \cdot u\)

- 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 \((e, \alpha)\) to local set
- update u: add delta to any remote replica

\[
\begin{align*}
\{5\} & \quad \{5\} \cup \{3\} = \{3, 5\} \quad \{3, 5\} \cup \{7\} = \{3, 5, 7\} \\
\{5\} & \quad \{3, 5\} \cup \{7\} = \{3, 5, 7\} \\
\{5\} & \quad \{5\} \cup \{7\} = \{5, 7\} \quad \{5, 7\} \cup \{3\} = \{3, 5, 7\} \\
\end{align*}
\]
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**
Conflict-free Replicated Data Types
Observed-Remove Set (CvRDT)

- **Payload**: \((A, R)\) - added/removed sets of \((\text{element, unique-token})\)
- **Operations**:
  - \(\text{add}(e)\): \(A := A \cup \{(e, \alpha)\}\)
  - \(\text{remove}(e)\): \(R := R \cup \{(e, -) \in A\}\) remove all unique elements observed
  - \(\text{lookup}(e)\): \(\exists (e, -) \in A \setminus R\)
  - \(\text{merge}(S, S')\): \((A \cup A', R \cup R')\)
- \(\{\text{true}\} \text{ add}(e) \parallel \text{remove}(e) \{e \in S\}\)
Conflict-free Replicated Data Types
Observed-Remove Set (CmRDT)

- **Payload:** 
  \[ S' = \{(e,\alpha), (e,\beta), (e,\gamma), \ldots\} \] where \( \alpha, \beta, \gamma, \ldots \) are unique tokens

- **Operations:** 
  \[ \text{add}(e): S := S \cup \{(e,\alpha)\} \] where \( \alpha \) is a fresh unique token
  \[ \text{lookup}(e): \exists \alpha: (e,\alpha) \in S \]
  \[ \text{remove}(e): R := \{(e,\alpha)\} \] at source, no tombstones
  \[ S := S \setminus R \]
  \[ \{\text{true}\} \text{ add}(e) \parallel \text{remove}(e) \{e \in S\} \]
Conflict-free Replicated Data Types
P-Counter (CvRDT)

• Payload:
  ▪ \( P = [\text{int}, \text{int}, \ldots] \)

• Operations:
  ▪ value(): \( \sum_i P[i] \)
  ▪ increment(): \( P[\text{MyID}]++ \)
  ▪ merge(S,S'): \( S \sqcup S' = [..., \text{max}(s.P[i], s'.P[i]), ...]_i \)

• 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'): $S \sqcup S' = ([…,\max(s.P[i],s'.P[i]),…],$
     $[…,\max(s.N[i],s'.N[i]),…])$

• Positive or negative
Conflict-free Replicated Data Types (CRDT)  
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 exactly once causally-ordered broadcast

• State-based:
  • Only require reliable broadcast
  • Communication overhead of shipping the whole state

• Delta State-based:
  • Small messages
  • Dissemination over unreliable communication channels
Conflict-free replicated data type (CRDT) (Text) Sequence

- Document = linear sequence of elements
- Unique position identifiers
  - Each element has a unique identifier
  - Identifier remains constant for the lifetime of the document
- Dense total order of identifiers consistent with element order:
  \[ \forall id_x, id_y : id_x < id_y \Rightarrow \exists id_z : id_x < id_z < id_y \]
- Real numbers require an infinite precision
- Different approaches for generating identifiers: Treedoc, Logoot
Logoot (*)

- Identifier:
  \[<p_1, s_1, h_1>, <p_2, s_2, h_2>, \ldots, <p_k, s_k, h_k>\]
  \(p_i\) integer
  \(s_i\) site identifier
  \(h_i\) logical clock at site \(s_i\)

\(<0, \text{NA}, \text{NA}>\>
\(<87,1,0>\>
\(<87,1,0>.<111,6,7>\>
\(<89,4,5>\>
\(<\text{MAX}, \text{NA}, \text{NA}>\>

(*) Stéphane Weiss, Pascal Urso and Pascal Molli. Logoot : a Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In ICDCS , Montreal, Quebec, Canada , June 2009

CSCW conference
March 19-23, 2011
Hangzhou,China
Site 9 wants to insert 5 lines between first 2 lines
- Take first components <1,1,1> and <4,2,1>
  - 2 identifiers: <2,9,h> and <3,9,h>
- Take first 2 components <1,1,1>.<5,2,4> and <4,2,1>.<2,4,6>
  - 25 identifiers
    - <1,1,1>.<{6-9},9,h>
    - <2,9,h>.<{1-9},9,h>
    - <3,9,h>.<{1-9},9,h>
    - <4,2,1>.<{1},9,h>
Choose 5 identifiers
Logoot

- Remote insertion: Insert(2,9,1>.<2,9,2>, “Computer Supported Cooperative Work”)
- Use binary search algorithm

CSCW conference
March 19-23, 2011
Hangzhou, China
### Logoot

- Remote deletion: \( \text{Delete}(<4,2,1>.<2,4,6>.<3,3,6>) \)
- Use binary search algorithm

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>1,1</td>
<td>5,2</td>
<td>4,6</td>
</tr>
<tr>
<td>2,9</td>
<td>1,2</td>
<td>9,2</td>
</tr>
<tr>
<td>4,2</td>
<td>2,4</td>
<td>6,3</td>
</tr>
<tr>
<td>6,3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

CSCW conference
Computer Supported Cooperative Work
March 19-23, 2011
Hangzhou, China
Operational Transformation (OT)

- Transforms non commuting operations to make them commute
- Genericity
- Time complexity
  - Average: $O(H \cdot c)$
  - Worst case: $O(H^2)$
- Difficult to write correct transformation functions
- State vectors used for detecting concurrency $\Rightarrow$ scalability limitations
- Not very suitable for large scale peer-to-peer collaboration
Conflict-free Replicated Data Types (CRDT)

- Time complexity
  - Average: $O(k \cdot \log(n))$
  - Worst case: $O(H \cdot \log(H))$

- No need for concurrency detection
- Identifiers storage cost
- New design for each data type
- Suitable for large-scale collaboration

<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1,2,1&gt;</td>
<td>c</td>
</tr>
<tr>
<td>&lt;1,2,2&gt;</td>
<td>o</td>
</tr>
<tr>
<td>&lt;2,1,2&gt;</td>
<td>n</td>
</tr>
<tr>
<td>&lt;3,1,3&gt;</td>
<td>c</td>
</tr>
<tr>
<td>&lt;3,1,3&gt;&lt;8,4,5&gt;</td>
<td>u</td>
</tr>
<tr>
<td>&lt;3,2,5&gt;</td>
<td>r</td>
</tr>
<tr>
<td>&lt;4,1,7&gt;</td>
<td>e</td>
</tr>
<tr>
<td>&lt;4,1,7&gt;&lt;9,2,6&gt;</td>
<td>n</td>
</tr>
<tr>
<td>&lt;7,2,8&gt;</td>
<td>c</td>
</tr>
<tr>
<td>&lt;9,1,7&gt;</td>
<td>y</td>
</tr>
<tr>
<td>&lt;10,2,8&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;12,3,1&gt;</td>
<td>c</td>
</tr>
<tr>
<td>&lt;12,3,1&gt;&lt;6,5,1&gt;</td>
<td>o</td>
</tr>
<tr>
<td>&lt;12,3,1&gt;&lt;7,8,2&gt;</td>
<td>n</td>
</tr>
<tr>
<td>&lt;12,3,1&gt;&lt;7,8,2&gt;&lt;12,3,5&gt;</td>
<td>t</td>
</tr>
<tr>
<td>&lt;12,3,1&gt;&lt;7,8,2&gt;&lt;13,3,6&gt;</td>
<td>r</td>
</tr>
<tr>
<td>&lt;12,3,1&gt;&lt;7,8,2&gt;&lt;14,3,7&gt;</td>
<td>l</td>
</tr>
</tbody>
</table>

\[ \text{ins(<3,2,5><13,1,7>, r)} \]

\[ \text{ins(<12,3,1><7,8,2><13,3,6><7,2,9>, o)} \]

| H: #ops | n: doc. size (non deleted chars.) | k: avg. size of Logoot identifier |
Conflict-free Replicated Data Types (CRDT) LogootSplit

LogootSplit identifiers

Base

- \( p_1 \)
- \( \ldots \)
- \( p_n \)
- \( site_id \)
- \( clock \)
- \( begin \)
- \( end \)

Interval

<table>
<thead>
<tr>
<th>Site ID</th>
<th>Clock</th>
<th>Begin</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1,0,5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,1,5,2,1,0,0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,1,6,16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Insert \( r \) between “concur” and “ency contr

- Insert \( o \) between “ency contr” and “l”

André, L. et al., “Supporting Adaptable Granularity of Changes for Massive Scale Collaborative Editing” CollaborateCom 2013
LogooSplit

Site 3

ABCDEF
2,1, [0.5]

insert XY between B and C

Site 4

ABCDEF
2,1, [0.5]
LogootSplit

Site 3

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDEF</td>
<td>2,1,</td>
<td>[0,1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*insert XY between B and C*

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABXY</td>
<td></td>
<td>CDEF</td>
</tr>
<tr>
<td>2,1,</td>
<td>[0,1]</td>
<td>2,1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2,5]</td>
</tr>
</tbody>
</table>
```

Site 4

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDEF</td>
<td>2,1,</td>
<td>[0,5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
**LogootSplit**

**Site 3**

ABCDEF
2,1,[0,5]

- **insert XY** between B and C

<table>
<thead>
<tr>
<th>AB</th>
<th>XY</th>
<th>CDEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1,[0,1]</td>
<td>2,1,1,3,1,[0,1]</td>
<td>2,1,[2,5]</td>
</tr>
</tbody>
</table>

**Site 4**

ABCDEF
2,1,[0,5]

- **insert ZT** between D and E

<table>
<thead>
<tr>
<th>ABCD</th>
<th>EF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1,[0,3]</td>
<td>2,1,[4,5]</td>
</tr>
</tbody>
</table>
**LogootSplit**

**Site 3**

- **ABCDEF**
  - 2,1,[0.5]

**Site 4**

- **ABCDEF**
  - 2,1,[0.5]

**Insertions**

- **Site 3**: Insert **XY** between B and C
- **Site 4**: Insert **ZT** between D and E

**Operations Sending**

- AB 2,1,[0,1]
- XY 2,1,[0,1]
- CDEF 2,1,[0,5]

- ABCD 2,1,[0,3]
- ZT 2,1,[3,4,1,0,1]
- EF 2,1,[4,5]

- AB 2,1,[0,1]
- XY 2,1,1,3,1,[0,1]
- CDEF 2,1,[2,5]
LogootSplit

Site 3

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>XY</td>
<td>CDEF</td>
<td></td>
</tr>
<tr>
<td>2,1,[0,1]</td>
<td>2,1,1,3,1,[0,1]</td>
<td>2,1,[0,5]</td>
<td></td>
</tr>
</tbody>
</table>

insert XY between B and C

Site 4

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td>ZT</td>
<td>EF</td>
<td></td>
</tr>
<tr>
<td>2,1,[0,3]</td>
<td>2,1,3,4,1,[0,1]</td>
<td>2,1,[4,5]</td>
<td></td>
</tr>
</tbody>
</table>

insert ZT between D and E

sending of operations
LogootSplit Performance Comparison

Random block (avg. 50 char.) insertion/deletion
First 80% insertions, then 20% insertions
Delays in MUTE
Delays in GoogleDocs