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 $H_s(n)$
- avant l’intégration $o_{p_1}$ $o_{p_2}$ ...
- $o_{p_i}$ $o_{p_{L_1}}$ $o_{p_{L_m}}$
- $o_{p_{i+1}}$

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

Intégration de $o_{p_{i+1}}$
- $o_{p_1}$ $o_{p_2}$ ...
- $o_{p_i}$ $o_{p_{L_1}}$ $o_{p_{L_m}}$
- $o_{p_{seq}}$

transpositions en avant des opérations locales

History $H_s(n)$
- après intégration $o_{p_1}$ $o_{p_2}$ ...
- $o_{p_i}$ $o_{p_{i+1}}$ $o_{p'_{L_1}}$ $o_{p'_{L_m}}$

opérations locales transposées en avant
So6- variation of SOCT4 algorithm(*)

```
Sync (log ,N_s) :-
    while ( (op_r = getOp (N_s+1)) ! = ∅) do
        for ( i = 0; i < log.size(); i ++)
            op_l = log [ i ];
            log [ i ] = T (op_l , op_r)
            op_r = T (op_r , op_l)
        endfor
        execute op_r
        N_s = N_s + 1
    endwhile

    for ( i = 0; i < log.size(); i ++)
        op_l = log [ i ];
        if send (op_l , N_s + 1) then
            N_s = N_s + 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

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></td>
</tr>
<tr>
<td></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
- \(op''3 = T(T(op3, op1), op2)\)
- \(op''4 = T(T(op4, op1'), op2')\)

**Site 2**

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

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

```
T(create(obj),_):-
    return create(obj)

T(change_color(obj1,c), del(obj2)):-
    if (obj1==obj2) then return id()
    else return change_color(obj1,c)

T(del(obj1), del(obj2)):-
    if (obj1==obj2) then return id()
    else return del(obj1)

T(del(obj1),change_color(obj2,c)):-
    return del(obj1)

T(change_color(obj1,c1),change(obj2,c2)):-
    if (obj1==obj2) then
        if(c1>c2) then return change_color(obj1,c1)
        else return id()
    else return change_color(obj1,c1)

T(id(),_):- return id()
T(op,id()):- return op
```
Site 1

\( \text{op1} = \text{del}(\text{obj1}) \)
\( \text{op2} = \text{change\_color}(\text{obj2}, \text{red}) \)
\( \text{op}''3 = \text{id()} \)
\( \text{op}''4 = \text{id()} \)

Site 2

\( \text{op3} = \text{change\_color}(\text{obj1}, \text{green}) \)
\( \text{op4} = \text{change\_color}(\text{obj2}, \text{blue}) \)

\( \text{red} > \text{blue} \)

\( \text{op}1 = \text{T}(\text{op1}, \text{op3}) = \text{del}(\text{obj1}) \)
\( \text{op}3 = \text{T}(\text{op3}, \text{op1}) = \text{id()} \)
\( \text{op}''1 = \text{T}(\text{op}''1, \text{op4}) = \text{del}(\text{obj1}) \)
\( \text{op}4 = \text{T}(\text{op4}, \text{op}''1) = \text{change\_color}(\text{obj2}, \text{blue}) \)
\( \text{op}2 = \text{T}(\text{op2}, \text{op}''3) = \text{change\_color}(\text{obj2}, \text{red}) \)
\( \text{op}''3 = \text{T}(\text{op}''3, \text{op2}) = \text{id()} \)
\( \text{op}''2 = \text{T}(\text{op}''2, \text{op}''4) = \text{change\_color}(\text{obj2}, \text{red}) \)
\( \text{op}''4 = \text{T}(\text{op}''4, \text{op}''2) = \text{id()} \)
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

- $\text{xform}(c,s)=\{c',s'\}$
- $\text{xform}(\text{del } x, \text{del } y)=
  \begin{cases} 
  \{\text{del } x-1, \text{del } y\} & \text{if } x>y \\
  \{\text{del } x, \text{del } y-1\} & \text{if } x<y \\
  \{\text{no-op, no-op}\} & \text{if } x=y 
  \end{cases}$

...
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;
}
\[ T(T(op3,op1),op2) \]

Outgoing: \([op1]\) at (0,0)

\[ T(op4,T(op2,T(op3,op1))) \]

Outgoing: \([op3]\) at (1,1)

\[ T(T(op3,op1),op2), T(op4,T(op2,T(op3,op1))) \]

Outgoing: \([op4]\) at (2,2)

\[ op'1 = T(op1, op3) \]

Outgoing: \([op1, op2]\) at (1,0)

\[ op'2 = T(T(op2,T(op3,op1)),op4) \]

Outgoing: \([op1, T(T(op2,T(op3,op1)),op4)]\) at (2,1)

\[ op'3 = T(T(op2,T(op3,op1)),op4) \]

Outgoing: \([op1, T(T(op2,T(op3,op1)),op4)]\) at (2,1)

\[ op'4 = T(T(op2,T(op3,op1)),op4) \]

Outgoing: \([op3]\) at (1,0)

\[ op'5 = T(T(op2,T(op3,op1)),op4) \]

Outgoing: \([op4]\) at (2,2)
Jupiter algorithm – generalisation n

Clients

msg

client 1

msg

client 2

msg

client 3

Algorithm changes at server side

\textbf{apply} \text{msg.op locally;}

\textbf{apply} \text{msg.op locally;}

\textbf{for} (c in client list) {
    \textbf{if} (c != \text{client})
    \textbf{send}(c, \text{msg});
}

Claudia Ignat, ignatcla@loria.fr
Jupiter algorithm

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

\[
\begin{align*}
\text{Site 1} & \quad \text{“abc”} \\
\text{Site 2} & \quad \text{“abc”} \\
\text{Site 3} & \quad \text{“abc”} \\
\end{align*}
\]

\[
\begin{align*}
op_1 &= \text{Insert}(2, x) \\
\text{“axbc”} \\
op_2 &= \text{Delete}(2, b) \\
\text{“ac”} \\
op_3 &= \text{Insert}(3, y) \\
\text{“abyc”} \\
\end{align*}
\]

\[
\begin{align*}
op_1’ &= \text{Insert}(2, x) \\
\text{“axyc”?} \\
op_3’ &= \text{Insert}(2, y) \\
\text{“ayxc”?} \\
\end{align*}
\]
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
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., $s \bullet m(s') = s \sqcup s'$(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 the following commands:
  - query q: returns the entire set
  - update u: adds a new element \((e, \alpha)\) to the local set
  - merge m: computes unions between the local set and the 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 commutativitiy 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
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:
  - add(e): \(A := A \cup \{(e, \alpha)\}\)
  - remove(e): \(R := R \cup \{(e, -)\in A\}\) remove all unique elements observed
  - lookup(e): \(\exists (e, -)\in A \setminus R\)
  - merge(S, S'): \((A \cup A', R \cup R')\)
- \{true\} add(e) \| 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
  - \( \text{lookup}(e) : \exists \alpha : (e, \alpha) \in S \)
  - \( \text{remove}(e) : R := \{(e, \alpha) | \exists \alpha (e, \alpha) \in S\} \) (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:**
  - $P = [\text{int, int, ...}]$

- **Operations:**
  - `value()`: $\sum_i P[i]$
  - `increment()`: $P[\text{MyID}]++$

  - `merge(S,S')`: $S \sqcup S' = [..., \max(s.P[i], s'.P[i]), ...]_i$

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

• Payload:
  - $P = \text{[int, int, ...]}$
  - $N = \text{[int, int, ...]}$

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

• 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. \textit{Logoot : a Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks}. In ICDCS, Montreal, Quebec, Canada, June 2009
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

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

| <0,NA,NA> |
|---|---|---|
| <1,1,1>.<5,2,4>.<4,6,2> |
| <4,2,1>.<2,4,6>.<3,3,6> |
| <6,3,4> |
| <MAX,NA,NA> |

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

---

| <0,NA,NA> |
|---|---|---|
| <1,1,1>.<5,2,4>.<4,6,2> |
| <2,9,1>.<2,9,2> |
| <4,2,1>.<2,4,6>.<3,3,6> |
| <6,3,4> |
| <MAX,NA,NA> |

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

Remote deletion: Delete(<4,2,1>.<2,4,6>.<3,3,6>)
Use binary search algorithm

Remote deletion: Delete(<4,2,1>.<2,4,6>.<3,3,6>)
Use binary search algorithm
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 ⇒ scalability limitations
- Not very suitable for large scale peer-to-peer collaboration

Site 1
- $op_1=\text{ins}(7,r)$
- $op_2=\text{ins}(18,o)$

Site 2
- $op_2=\text{ins}(17,o)$
- $op_1=\text{ins}(7,r)$
Conflict-free Replicated Data Types (CRDT)

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

  $H$: #ops
  $n$: doc. size (non deleted chars.)
  $k$: avg. size of Logoot identifier

- **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

LogootSplit identifiers

Base

| p₁ | … | pₙ | site_id | clock | begin | end |

Interval

| 1,1,[0,16] | concurrency 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 |

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

Site 3

\begin{itemize}
  \item ABCDEF
  \item 2,1, [0,5]
\end{itemize}

\[ \text{insert XY between B and C} \]

Site 4

\begin{itemize}
  \item ABCDEF
  \item 2,1, [0,5]
\end{itemize}
LogootSplit

Site 3

ABCDEF
2,1,[0,5]

insert XY between B and C

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

Site 4

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

**Site 3**

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

  insert XY between B and C

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

**Site 4**

- ABCD
  - 2,1,[0,3]

  insert ZT between D and E

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

Site 3

ABCDEF
2,1,[0,5]

insert XY between B and C

Site 4

ABCDEF
2,1,[0,5]

insert ZT between D and E

sending of operations

AB
2,1,[0,1]

XY
2,1,1,3,1,[0,1]

CDEF
2,1,[2,5]

ABC
2,1,1,3,1,[0,1]

CDEF
2,1,[2,5]

ABCD
2,1,[0,3]

ZT
2,1,3,4,1,[0,1]

EF
2,1,[4,5]

ZT
2,1,3,4,1,[0,1]

EF
2,1,[4,5]
LogootSplit

Site 3

ABCDEF
2,1,[0,5]

insert XY between B and C

Site 4

ABCDEF
2,1,[0,5]

insert ZT between D and E

Sending of operations

AB
2,1,[0,1]

XY
2,1,1,3,1,[0,1]

CDEF
2,1,[2,5]

ABCD
2,1,[0,3]

ZT
2,1,3,4,1,[0,1]

EF
2,1,[4,5]

insert XY between B and C

insert ZT between D and E

AB
2,1,[0,1]

XY
2,1,1,3,1,[0,1]

CD
2,1,[2,3]

ZT
2,1,3,4,1,[0,1]

CD
2,1,[2,3]

XY
2,1,1,3,1,[0,1]

EF
2,1,[4,5]
LogootSplit Performance Comparison

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

![Graph showing delays in MUTE with number of users on the x-axis and delays in seconds on the y-axis. The graph includes a trend line indicating the delay trend as the number of users increases.]
Delays in GoogleDocs