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

• $a \rightarrow b$ ($a$ happens before $b$)

• Two situations:
  - If $a$ and $b$ are events in the same process and $a$ occurs before $b$, then $a \rightarrow b$
  - 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 $a \rightarrow b$. A message cannot be received before or at the same time it is sent

• If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

• If neither $a \rightarrow b$ nor $b \rightarrow a$ 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 $< \text{time of the arrived message}$, then receiver forwards its clock to $1 + \text{sending time}$
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 $a \rightarrow b$
  ▪ $a \parallel b$ 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 occurred 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 $\forall i \ V[i]=W[i]$
  - $V<W$ iff $\forall i \ V[i] \leq W[i]$ and $\exists i \ V[i]<W[i]$
  - $[1,2,0] < [3,2,1]$
  - $[0,1,1] \not< [1,0,1]$
Vector timestamps – computation rules

• Process Pi
  - Initialisation: $\forall k \ Vi[k]=0$
  - Local event: $Vi[i]= Vi[i]+1$
  - Sending message m : $Vi[i]= Vi[i]+1$, then send $(m,Vi)$
  - Receiving message $(m,Vj)$:
    - $\forall k \ Vi[k]=\max(Vi[k], \ Vj[k])$
    - $Vi[i]=Vi[i]+1$
Vector timestamps – example

- \( p_1 \):
  - a: \([1,0,0]\)
  - b: \([2,0,0]\)
  - c: \([3,0,0]\)

- \( p_2 \):
  - d: \([0,1,0]\)
  - e: \([2,2,0]\)
  - f: \([2,3,0]\)

- \( p_3 \):
  - g: \([0,0,1]\)
  - h: \([0,0,2]\)
  - i: \([2,3,3]\)
State vector
State vector– computation rules

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

Duplicated databases (Thomas Write Rule 1975)

DBMP\(_1\)  DBMP\(_2\)  DBMP\(_3\)

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)

- 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
  - \((T_1,D_1) > (T_2,D_2)\) iff \((T_1 > T_2)\) or \((T_1 = T_2\) and \(D_1 > D_2)\)
- If \(D_1 = D_2\) and \(T_1 = T_2\), 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

```
Database
  └── new(x,5) ─── (x,5,(10h,1))
  └── set(x,8) ─── (x,8,(10h02,1))
      └── (x,9,(10h03,2))
          └── (10h02,1)<(10h03,2)

DBMP₁
  └── (x,5,(10h,1))

DBMP₂
  └── (x,5,(10h,1))
      └── set(x,9) ─── (x,9,(10h03,2))
          └── (x,9,(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

DBMP_1
new(x,2)

DBMP_2
set(x,3)
(x,2,...)

DBMP_3
(x,3,...)
ignored
Deletion

- 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
(x,3,f,(10h,2))
del(x)
(x,3,t,(10h01,1))

DBMP_2
(x,3,f,(10h,2))
set(x,5)
(x,5,f,(10h02,2))

DBMP_3
(x,3,f,(10h,2))
(x,5,f,(10h02,2))

DBMP_4
(x,3,f,(10h,2))
(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

\[ \text{DBMP}_1 \quad \text{DBMP}_2 \]

- delete(x)
  \[(x,3,f,(10h,2))\]
- set(x,5)
  \[(x,3,t,(10h01,1))\]
  \[(x,3,t,(10h01,1))\]

DBMP1 cannot distinguish in which of the two cases DBMP2 is

- delete(x)
  \[(x,3,f,(10h,2))\]
- set(x,5)
  \[(x,5,f,(10h02,2))\]
  \[(x,3,t,(10h01,1))\]

• Solution: Associate to an entry the creation timestamp

Recreation!

Divergence!
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

DBMP₁

delete(x)

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

set(x,5)

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

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

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

Same creation time => delete

DBMP₂

set(x,5)

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

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

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

Same creation time => delete
Tombstones

DBMP\(_1\)

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

**delete**(x)

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

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

DBMP\(_2\)

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

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

**set**(x, 5)

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

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 $\leq$ 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=[]{(1h,1)}
OT=[]

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

DBMP_3

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

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

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

new(y)

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

new(z)

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

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

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

(z,3,f,(3h,3),(3h,3))
Garbage collection

DBMP₁
LM=[(2h,2),(3h,3)]
OT=[]
delete(z)
LM=[(2h,2),(3h,3)]
OT=[(3h,2)]
LM=[(2h,2),(3h,3)]
OT=[(3h,2),(2h,3)]
LM=[(4h,1),(3h,3)]
OT=[(2h,3)]
LM=[(4h,1),(3h,3)]
OT=[(3h,2),(2h,3)]
LM=[(4h,1),(3h,3)]
OT=[(3h,2)]
LM=[(4h,1),(3h,3)]
OT=[(3h,2)]
LM=[(4h,1),(2h,2)]
OT=[(3h,2)]
LM=[(4h,1),(2h,2)]
OT=[(3h,2)]
LM=[(1h,1),(3h,3)]
OT=[]
LM=[(1h,1),(2h,2)]
OT=[]
LM=[(1h,1),(3h,3)]
OT=[]
LM=[(1h,1),(3h,3)]
OT=[]
Garbage collection

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

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

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

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

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

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

• 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

\texttt{op1=ins(1,y)}

\texttt{op2=del(1)}

Site 2

\texttt{op1=ins(1,y)}

\texttt{op2=del(1)}

Site 3

\texttt{op1=ins(1,y)}

\texttt{op2=del(1)}
Causality

Site 1

\[ [0,0,0] \]

“X”

op1=ins(1,y)

[1,0,0]

“yX”

op2=del(1)

[1,1,0]

“X”

Site 2

\[ [0,0,0] \]

“X”

op1=ins(1,y)

[1,0,0]

“yX”

Site 3

\[ [0,0,0] \]

“X”

op2=del(1)

[1,1,0]

“X”

op1=del(1)

[1,0,0]

“y”

[0,0,0]

delayed
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.

Intention violation

Site 1
"concurrency control"

op1=ins(7,r)

"concurrency control"

op2=ins(17,o)

Site 2
"concurrency control"

"concurrency control"
**Intention violation + divergence**

**Site 1**

- \( op1 = \text{ins}(7,r) \)
- \( \text{“concurrency contrl”} \)

**Site 2**

- \( op2 = \text{ins}(17,o) \)
- \( \text{“concurrency control”} \)

**Connections**

- \( \text{op}1 = \text{ins}(7,r) \) to \( \text{Site 2} \)
- \( \text{op}2 = \text{ins}(17,o) \) to \( \text{Site 1} \)
Intention preservation

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

op1 = ins(7,r)

Site 1

"concurrency control"

op2 = ins(17,o)

Site 2

"concurrency control"

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

op1 = ins(7,r)

Site 1

"concurrency control"

op2 = ins(17,o)

Site 2

"concurrency control"

op2 = ins(17,o)

op'2 = ins(18,o)
Example transformation functions

\[ T(\text{Ins}(p1,c1), \text{Ins}(p2,c2)) :- \]
\[
\begin{align*}
\text{if} & \ (p1 < p2) \ \text{return} \ \text{Ins}(p1,c1) \\
\text{else} & \ \text{return} \ \text{Ins}(p1+1,c1)
\end{align*}
\]

\[ T(\text{Ins}(p1,c1), \text{Del}(p2)) :- \]
\[
\begin{align*}
\text{if} & \ (p1 \leq p2) \ \text{return} \ \text{Ins}(p1,c1) \\
\text{else} & \ \text{return} \ \text{Ins}(p1-1,c1) \\
\text{endif}
\end{align*}
\]

\[ T(\text{Del}(p1), \text{Ins}(p2,c2)) :- \]
\[
\begin{align*}
\text{if} & \ (p1 < p2) \ \text{return} \ \text{Del}(p1) \\
\text{else} & \ \text{return} \ \text{Del}(p1+1)
\end{align*}
\]

\[ T(\text{Del}(p1), \text{Del}(p2)) :- \]
\[
\begin{align*}
\text{if} & \ (p1 < p2) \ \text{return} \ \text{Del}(p1) \\
\text{else if} & \ (p1 > p2) \ \text{return} \ \text{Del}(p1-1) \\
\text{else} & \ \text{return} \ \text{Id}()
\end{align*}
\]
Convergence but no intention preservation

Thomas Write Rule

Site 1

\[(s, "AB", f, (0h, 1), (0h, 1))\]

\[
\text{op1=}\text{set}(s, "AXB")
\]

\[(s, "AXB", f, (0h, 1), (9h01, 1))\]

Site 2

\[(s, "AB", f, (0h, 1), (0h, 1))\]

\[(s, "AXB", f, (0h, 1), (9h01, 1))\]

Site 3

\[(s, "AB", f, (0h, 1), (0h, 1))\]

\[
\text{op2=}\text{set}(s, "AYB")
\]

\[(s, "AYB", f, (0h, 1), (9h02, 3))\]
Convergence – TP1 property

- $T(\text{op2}: \text{operation}, \text{op1}: \text{operation}) = \text{op’2}$
  - op1 and op2 concurrent, defined on a state $S$
  - op’2 same effects as op2, defined on $S.\text{op1}$

\[
[\text{TP1}] \quad \text{op1} \circ T(\text{op2}, \text{op1}) \equiv \text{op2} \circ T(\text{op1},\text{op2})
\]
Convergence – TP2 property

\[ TP2 \quad T(\text{op}3, \text{op}1 \circ T(\text{op}2, \text{op}1)) = T(\text{op}3, \text{op}2 \circ T(\text{op}1, \text{op}2)) \]
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

```
op1 = ins(5,p)
```

Site 2

```
op2 = del(5)
```

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

```
op’1 = ins(5,p)
```

```
op2 = T(op2, op1) = del(6)
op”2 = T(op’2, op3) = del(7)
op’1 = T(op1, op2) = ins(5)
```

```
op3 = ins(6,h)
```

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

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

```
op”2 = del(7)
```

```
op’2 = ins(5)
```

```
```
```
Partial concurrency

Site 1

```
op1=ins(5,p)  "telefone"
```

```
op2=del(5)  "telefone"
```

```
op3=ins(6,h)  "telephfone"
```

```
op'2=del(7)  "telephone"
```

Site 2

```
op2=del(5)  "teleone"
```

```
transpose
bk
```

```
op'1=ins(5,p)  "telepone"
```

```
transpose
bk
```

```
op’3=T(op3,op’2) =ins(6,h)  "telepone"
```

```
transpose
bk
```

```
op’2=T(op2,op1)=del(6)  "telefone"
```

```
op1=ins(5,p)  "telefone"
```

```
transpose
bk
```

```
transpose
bk
```

```
transpose
bk
```
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

\[ T(\text{Ins}(p_1, c_1), \text{Ins}(p_2, c_2)) : \]
\[
\begin{cases}
    \text{if } (p_1 < p_2) & \text{return } \text{Ins}(p_1, c_1) \\
    \text{else} & \text{return } \text{Ins}(p_1 + 1, c_1)
\end{cases}
\]

\[ T(\text{Ins}(p_1, c_1), \text{Del}(p_2)) : \]
\[
\begin{cases}
    \text{if } (p_1 \leq p_2) & \text{return } \text{Ins}(p_1, c_1) \\
    \text{else} & \text{return } \text{Ins}(p_1 - 1, c_1) \\
    \text{endif}
\end{cases}
\]

\[ T(\text{Del}(p_1), \text{Ins}(p_2, c_2)) : \]
\[
\begin{cases}
    \text{if } (p_1 < p_2) & \text{return } \text{Del}(p_1) \\
    \text{else return } \text{Del}(p_1 + 1)
\end{cases}
\]

\[ T(\text{Del}(p_1), \text{Del}(p_2)) : \]
\[
\begin{cases}
    \text{if } (p_1 < p_2) & \text{return } \text{Del}(p_1) \\
    \text{else if } (p_1 > p_2) & \text{return } \text{Del}(p_1 - 1) \\
    \text{else return } \text{Id}()
\end{cases}
\]

TP1 not respected!
**Ressel transformation functions (*)**

\[
T(\text{Ins}(p1,c1,u1), \text{Ins}(p2,c2,u2)) :\]
\[
\text{if } ((p1<p2) \text{ or } (p1=p2 \text{ and } u1<u2)) \text{ return } \text{Ins}(p1,c1,u1)
\]
\[
\text{else return } \text{Ins}(p1+1,c1,u1)
\]

\[
T(\text{Ins}(p1,c1,u1), \text{Del}(p2,u2)) :\]
\[
\text{if } (p1\leq p2) \text{ return } \text{Ins}(p1,c1,u1)
\]
\[
\text{else return } \text{Ins}(p1-1,c1,u1)
\]
\[
\text{endif}
\]

\[
T(\text{Del}(p1,u1), \text{Ins}(p2,c2,u2)) :\]
\[
\text{if } (p1<p2) \text{ return } \text{Del}(p1,u1)
\]
\[
\text{else return } \text{Del}(p1+1,u1)
\]

\[
T(\text{Del}(p1,u1), \text{Del}(p2,u2)) :\]
\[
\text{if } (p1<p2) \text{ return } \text{Del}(p1,u1)
\]
\[
\text{else if } (p1>p2) \text{ return } \text{Del}(p1-1,u1)
\]
\[
\text{else return } Id()
\]

TP1 ok, but not TP2 !
**Suleiman transformation functions (•)**

\[ \text{Ins}(p, c, a, b) \]

- \( b \) – operations that have concurrently deleted a character before character \( c \)
- \( a \) – operations that have concurrently deleted a character after character \( c \)

Two concurrent \( \text{ins}(p, c_1, a_1, b_1) \) and \( \text{ins}(p, c_2, a_2, b_2) \)

If \( b_1 \cap a_2 \neq \emptyset \), at generation \( p_2 < p_1 \)

If \( a_1 \cap b_2 \neq \emptyset \), at generation \( p_1 < p_2 \)

If \( b_1 \cap a_2 = a_1 \cap b_2 = \emptyset \), at generation \( p_1 = p_2 \)

Suleiman transformation functions

\[ T(\text{Ins}(p_1,c_1,a_1,b_1), \text{Ins}(p_2,c_2,a_2,b_2)) :- \]

\[ \text{if } (p_1 > p_2) \text{ then return } \text{Ins}(p_1+1,c_1,a_1,b_1); \]

\[ \text{else if } (p_1 < p_2) \text{ then return } \text{Ins}(p_1,c_1,a_1,b_1); \]

\[ \text{else if } (p_1 = p_2) \text{ then} \]

\[ \text{if } (b_1 \cap a_2 \neq \emptyset) \text{ then return } \text{Ins}(p_1+1,c_1,a_1,b_1); \]

\[ \text{else if } (a_1 \cap b_2 \neq \emptyset) \text{ then return } \text{Ins}(p_1,c_1,a_1,b_1); \]

\[ \text{else if } (\text{code}(c_1) > \text{code}(c_2)) \text{ then return } \text{Ins}(p_1,c_1,a_1,b_1); \]

\[ \text{else if } (\text{code}(c_1) < \text{code}(c_2)) \text{ then return } \text{Ins}(p_1+1,c_1,a_1,b_1); \]

\[ \text{else return } \text{id}(\text{Ins}(p_1,c_1,a_1,b_1)); \]
Suleiman transformation functions

\[ T(\text{Ins}(p_1,c_1,a_1,b_1), \text{Del}(p_2)) :\]
\[ \text{if } (p_1 > p_2) \text{ return } \text{Ins}(p_1-1,c_1,b_1+\text{Del}(p_2),a_1) \]
\[ \text{else return } \text{Ins}(p_1,c_1,b_1,a_1+\text{Del}(p_2)) \]
\[ \text{endif} \]

\[ T(\text{Del}(p_1), \text{Del}(p_2)) :\]
\[ \text{if } (p_1 < p_2) \text{ return } \text{Del}(p_1) \]
\[ \text{else if } (p_1 > p_2) \text{ return } \text{Del}(p_1-1) \]
\[ \text{else return } \text{Id}(\text{Del}(p_1)) \]

\[ T(\text{Del}(p_1), \text{Ins}(p_2,c_2,a_2,b_2)) :\]
\[ \text{if } (p_1 < p_2) \text{ return } \text{Del}(p_1) \]
\[ \text{else return } \text{Del}(p_1+1) \]
Suleiman transformation functions

\[
del(3) \\ \Rightarrow del(3) \\
ins(3, x, \emptyset, \emptyset) \ins(3, x, \emptyset, \{\text{del}(3)\}) \ins(3, y, \{\text{del}(3)\}, \emptyset) \\
\text{id}() \\
ins(5, x, \emptyset, \emptyset)
\]
False-tie problem

Site 1
“abc”

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

“axbc”

Site 2
“abc”

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

“ac”

Site 3
“abc”

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

“abyyc”

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

Site 1
“abc”

Site 2
“abc”

Site 3
“abc”

\[ \text{op'}_3 = \text{Insert}(2, y) \]

“axyc”? “ayxc”?
TTF (Tombstone Transformation Functions) Approach (*)

- Keep “tombstones” of deleted elements

Tombstone Transformation Functions

- \( T(\text{Insert}(p_1, e_1, s_1), \text{Insert}(p_2, e_2, s_2)) \)\
  \[
  \begin{align*}
  &\text{if}(p_1 < p_2) \text{ return } \text{Insert}(p_1, e_1, s_1) \\
  &\text{else if}(p_1 = p_2 \text{ and } s_1 < s_2) \text{ return } \text{Insert}(p_1, e_1, s_1) \\
  &\text{else return } \text{Insert}(p_1+1, e_1, s_1)
  \end{align*}
  \]

- \( T(\text{Insert}(p_1, e_1, s_1), \text{Delete}(p_2, e_2, s_2)) \)\
  \[
  \text{return } \text{Insert}(p_1, e_1, s_1)
  \]

- \( T(\text{Delete}(p_1, s_1), \text{Insert}(p_2, s_2)) \)\
  \[
  \begin{align*}
  &\text{if}(p_1 < p_2) \text{ return } \text{Delete}(p_1, s_1) \\
  &\text{else return } \text{Delete}(p_1+1, s_1)
  \end{align*}
  \]

- \( T(\text{Delete}(p_1, s_1), \text{Delete}(p_2, s_2)) \)\
  \[
  \text{return } \text{Delete}(p_1, s_1)
  \]
• Compacted model = sequence of (character, abs_pos)
Delta storage model

- 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