Chapter 7: Lazy Replication

Update Propagation Protocols (1)

- Alternatives to update replicas: synchronous vs. asynchronous:
  - synchronous – within transaction (eager replication),
  - asynchronous – separate transaction to update replicas (lazy replication).
- Synchronous updates of replicas: no scalability, in particular if each peer issues transactions. Each transaction must lock quorum or primary copy.

Update Propagation Protocols (2)

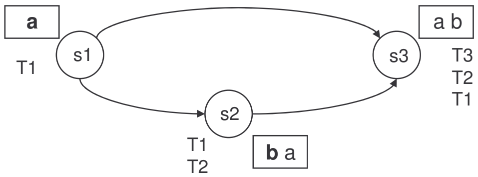
- Furthermore:
  - With synchronous updates, we must wait for the slowest one. Illustration.
  - Failure of nodes – similar problem.

Update Propagation Protocols (3)

- Lazy replication – underlying idea:
  - If update has been successful on one node, it will eventually be successful on all other nodes as well. Illustrated on subsequent slide.
  - I.e., transaction may already commit without having updated all replicas.

Introduction
DAG(WT)
DAG(T)
BackEdge

Lazy Replication & Primary-Copy Technique – Illustration



- Three transactions: T_1 on Site 1 updates a.
  T_2 on Site 2 reads a and writes b.
  T_3 on Site 3 reads a and b.
- Lazy propagation of updates – possible sequence of executions:
  - on Site 2: $w_1[a]\ r_2[a]\ w_2[b]$
  - on Site 3: $w_2[b]\ r_3[a]\ r_3[b]\ w_1[a]$

Introduction
DAG(WT)
DAG(T)
BackEdge

Update Propagation Protocols (4)

- Eager vs. lazy:
  - Lazy – performance tends to be better (but not by orders of magnitude, depending on the distribution scheme of the replicas),
  - serializability typically not guaranteed with lazy.

Introduction
DAG(WT)
DAG(T)
BackEdge

Update Propagation Protocols (5)

- Our topic in what follows:
  lazy protocols guaranteeing serializability.
  1. Protocols making assumptions regarding the arrangement of replicas.
  2. Hybrid protocol without such assumptions that is lazy whenever possible.

Introduction
DAG(WT)
DAG(T)
BackEdge

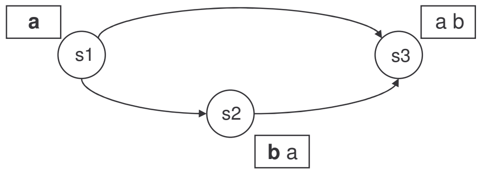
System Model/Assumptions

- Each data object has a *primary site*.
  ⇒ *primary copy*,
  secondary copies/replicas.
- Transaction has ‚originating site‘.
- Transaction can only modify data objects whose primary site = originating site; but may read any data object.
- Nodes (sites) use 2PL.
- Network is reliable;
  delivery of messages in FIFO order.
- *Primary subtransaction*,
  secondary subtransaction.

Copy Graph

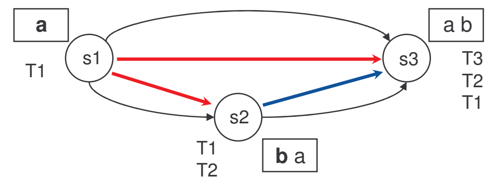
Copy Graph:

- nodes ≡ sites.
- Edge from s_i to s_j iff primary copy of a data object is on s_i, and ∃ secondary copy on s_j.
- Example: three sites, two data objects a and b.



- *Backedges* := set of edges s.t. deletion of such edges makes the graph cycle-free.

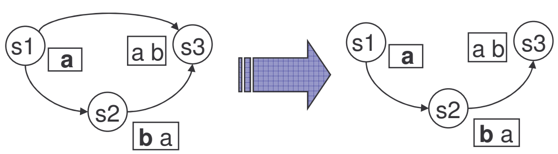
Example of Non-serializable Execution



- Three transactions: T_1 on Site 1 updates a. T_2 on Site 2 reads a and writes b. T_3 on Site 3 reads a and b.
- Lazy propagation of updates – possible sequence of executions:
  - on Site 2: $w_1[a]\ r_2[a]\ w_2[b]$
  - on Site 3: $w_2[b]\ r_3[a]\ r_3[b]\ w_1[a]$

DAG(WT) Protocol (1)

- „DAG without Timestamps“
- Prerequisite: copy graph is cycle-free.
- Generate Tree T from copy graph: s_i child of s_j. ⇒ s_i successor of s_j in T.
- Example:



DAG(WT) Protocol (2)

- Node forwards update transactions to children in T.
- Commit order
  = order in which transactions arrive at node
  = order in which transactions are forwarded to children.

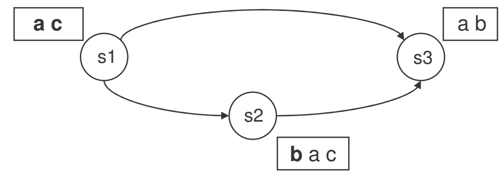
DAG(WT) Protocol (3)

- Point that is still open:
  - Local deadlocks feasible, i.e., transaction does not necessarily commit.
  - Local deadlock feasible because of
    - conflict of T with transaction T_x that has not occurred at other sites. Illustration on subsequent slide.
    - Or interaction with transaction that has already occurred at other sites. No handshaking, only commit order is given.
  - But local transaction must commit.
  - Thus, victim selection policy should be fair, e.g.: last transaction.

DAG(WT) Protocol (4)

Introduction
DAG(WT)
DAG(T)
BackEdge

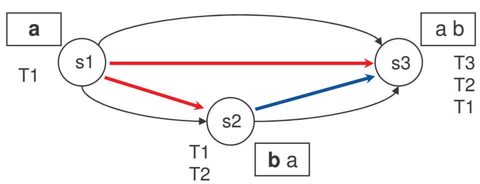
- Point that is still open (cont.):
  - Local deadlocks – illustration:



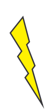
- T_1: $r_1(c)$ $w_1(c)$ $w_1(a)$, T_2: $r_2(a)$ $r_2(c)$ $w_2(b)$
- Possible execution at s2: $w_1(c)$ $r_2(a)$ $r_2(c)$ $w_1(a)$

Example of Non-serializable Execution

Introduction
DAG(WT)
DAG(T)
BackEdge



- Three transactions: T_1 on Site 1 updates a. T_2 on Site 2 reads a and writes b. T_3 on Site 3 reads a and b.
- Lazy propagation of updates – possible sequence of executions:
  - on Site 2: $w_1[a]$ $r_2[a]$ $w_2[b]$
  - on Site 3: $w_2[b]$ $r_3[a]$ $r_3[b]$ $w_1[a]$

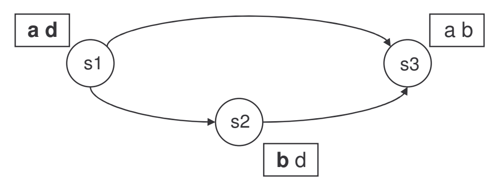


Z

DAG(WT) Protocol – Observation

Introduction
DAG(WT)
DAG(T)
BackEdge

- Transaction is routed to nodes that are not relevant.



- Delays.

DAG(T) Protocol

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- „DAG with Timestamps“
- Propagate update transactions along edges of copy graph.
- Primary subtransactions have timestamp that specify execution order.

- Outline of the following:
  - Structure of timestamp (total order necessary),
  - protocol itself.
- Structure of timestamps is interesting, since their generation is decentralized.

Timestamps (1)

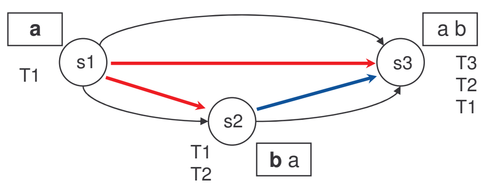
Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- In what follows:
  - auxiliary notion *local timestamp counter*,
  - auxiliary notion *timestamp of a site*,
  - *timestamp of a transaction*.

- Acyclicity results in total order of sites; $s_i < s_{i+k}$
- auxiliary construct –
  tuple corresponding to Node s_i : (s_i, LTS_i)
  LTS = *Local Timestamp Counter*;
  counts the primary subtransactions
  that have committed there.

Local Timestamp Counters – Example

Introduction
DAG(WT)
DAG(T)
BackEdge



(s1, 0), (s2, 0), (s3, 0)

(s1, 1)

(s2, 1)

Timestamps (2)

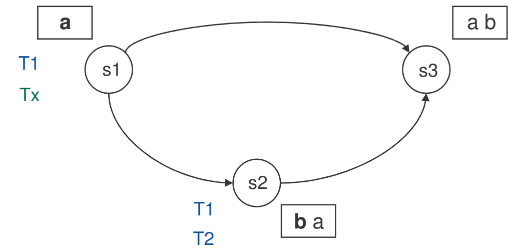
Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- *Timestamp of a site s_i:*
  - vector of tuples,
  - a tuple for s_i
    and further tuples for predecessors of s_i
    in copy graph.
  - Important:
    tuple in vector ordered by sites.
- Timestamp of a site reflects
  how many primary subtransactions
  and how many secondary subtransactions
  have committed there.

Timestamps (3)

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

Example:



1

Site	Timestamp
s_1	<(s_1, 0)>
s_2	<(s_1, 0), (s_2, 0)>
...	...

2

Site	Timestamp
s_1	<(s_1,1)>
s_2	<(s_1,0), (s_2,0)>
...	...

3

Site	Timestamp
s_1	<(s_1, 1)>
s_2	<(s_1, 1), (s_2, 0)>
...	...

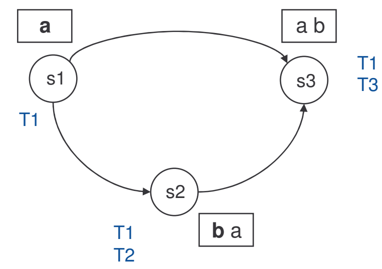
4

Site	Timestamp
s_1	<(s_1, 1)>
s_2	<(s_1, 1), (s_2, 1)>
...	...

Timestamps (4)

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- *Timestamp $TS(T_i)$ of a transaction*: timestamp of the site of the primary transaction immediately after the point of time of commit.
- Example:



T1: < (s_1, 1)>
T2: <(s_1, 1), (s_2, 1)>
T3: <(s_1, 1), (s_3, 1)>

Timestamps (5)

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- Order of timestamps shall reflect commit order.
- Lexicographic ordering < of timestamps:
  $TS_1 < TS_2$:⇔
  - TS_1 is prefix of TS_2, or
  - TS_1=X (s_i, LTS_i) Y_1, TS_2=X (s_j, LTS_j) Y_2, and
    1. $s_i > s_j$, or
    2. $s_i = s_j$, and $LTS_i < LTS_j$.

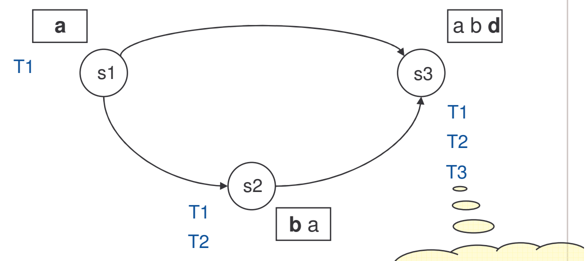
Timestamps – Explanation (1)

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- Order of timestamps shall reflect commit order.
- $TS_1 < TS_2 \Leftarrow TS_1$ is prefix of TS_2.
  Example: <(s_1, 1)> describes state before <(s_1, 1), (s_2, 1)>.
- $TS_1 < TS_2 \Leftarrow TS_1$=X ($s_i$, LTS_i) Y_1, TS_2=X (s_j, LTS_j) Y_2, and $s_i = s_j$, und $LTS_i < LTS_j$.
  Example: <(s_1, 1)> describes state before <(s_1, 2)>.

Timestamps – Example, Continued (1)

Example:



1

Site	Timestamp
s_1	$<(s_1, 0)>$
s_2	$<(s_1, 0), (s_2, 0)>$
...	...

2

Site	Timestamp
s_1	$<(s_1,1)>$
s_2	$<(s_1,0), (s_2,0)>$
...	...

3

Site	Timestamp
s_1	$<(s_1, 1)>$
s_2	$<(s_1, 1), (s_2, 0)>$
...	...

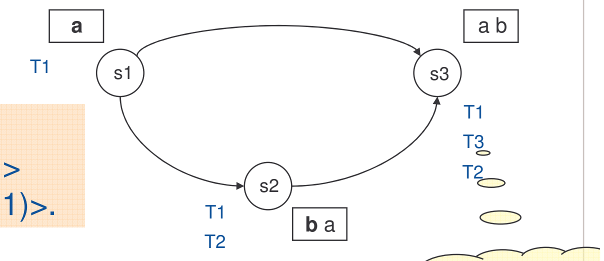
4

Site	Timestamp
s_1	$<(s_1, 1)>$
s_2	$<(s_1, 1), (s_2, 1)>$
...	...

Timestamps – Example, Continued (2)

Example:

Hence,
$<(s_1, 1), (s_2, 1)>$
$> <(s_1, 1), (s_3, 1)>$.



1

Site	Timestamp
s_1	$<(s_1, 0)>$
s_2	$<(s_1, 0), (s_2, 0)>$
...	...

2

Site	Timestamp
s_1	$<(s_1,1)>$
s_2	$<(s_1,0), (s_2,0)>$
...	...

3

Site	Timestamp
s_1	$<(s_1, 1)>$
s_2	$<(s_1, 1), (s_2, 0)>$
...	...

4

Site	Timestamp
s_1	$<(s_1, 1)>$
s_2	$<(s_1, 1), (s_2, 1)>$
...	...

Data Structures

Data structure for each node:

- timestamp vector of the node
  (= timestamp of the last committed secondary subtransaction + tuple of the node),
- waiting queues,
  one for each predecessor.

Primary Transaction

Primary transaction commits at Node s_i:

1. increment LTS_i (Local Timestamp Counter),
2. $TS(T_i) := TS(s_i)$
3. Send secondary (sub-)transaction to children of s_i.

- Timestamp of a site reflects how many primary subtransactions and how many secondary subtransactions have committed there.

Secondary Transaction (1)

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- Assumption:
  only one secondary transaction at a time.
- One waiting queue for each direct predecessor of the site in the copy graph.
- Choose transaction from queues with minimal timestamp.
- There must be at least one transaction in each queue before computing the min timestamp.
- Idle nodes should commit dummy transactions from time to time.

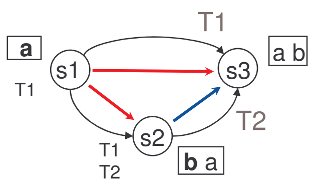
Secondary Transaction (2)

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge

- After commit: $TS(s_i) := TS(T_i)(s_i, LTS_i)$
- How do we know that next transaction in commit order is not being propagated through the network?
  Corresponding queue would be empty, furthermore: FIFO order.

Continuation of Example

Introduction
DAG(WT)
DAG(T)
- Introduction
- Timestamps
- Protocol
BackEdge



- T_1 has Timestamp $(s_1, 1)$,
- T_2 has Timestamp $(s_1, 1), (s_2, 1)$.
  (When T_1 commits on s_2,
  the site timestamp is set to $(s_1, 1), (s_2, 0)$.)
- Site s_3:
  timestamp of T_1 is prefix of the one of T_2.
  $\Rightarrow$ T_1 is executed there before T_2.

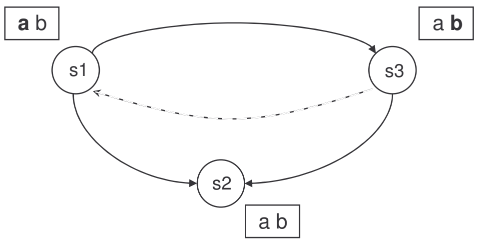
BackEdge Protocol – Motivation

Introduction
DAG(WT)
DAG(T)
BackEdge

- Copy graph must be acyclic for DAG(WT) and DAG(T) protocols.
- Example:
  - two sites s_1 and s_2.
  - s_1 holds primary copy of a and copy of b, s_2 vice versa.
  - T_1 at Node s_1 reads b and updates a,
    T_2 at Node s_2 reads a and updates b.
  - Both transactions execute concurrently and commit. Illustration.
  - No serializability.

BackEdge Protocol – Overview

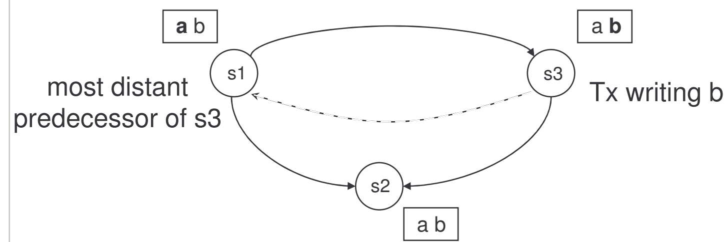
- Hybrid: for some replicas eager updates, for other ones lazy.
- Can be described both as extension of DAG(WT) and DAG(T).
- G_{dag} – results from G by removing backedges.



Introduction
DAG(WT)
DAG(T)
BackEdge

BackEdge Protocol – Terminology (1)

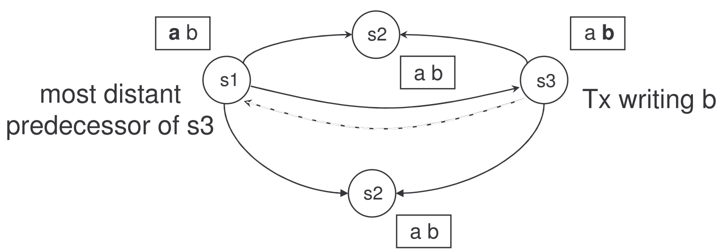
- Backedge from s_i to s_j.
  $\Rightarrow$ s_j is predecessor of s_i in G_{dag}.
- T_i – primary subtransaction with node s_i.
- „Backedge subtransactions“: transactions S_1, ..., S_j at predecessor s_{i1}, ..., s_{ij} of s_i in T. s_{i1} „most distant“ of s_i etc.



Introduction
DAG(WT)
DAG(T)
BackEdge

BackEdge Protocol – Terminology (2)

- Illustration:

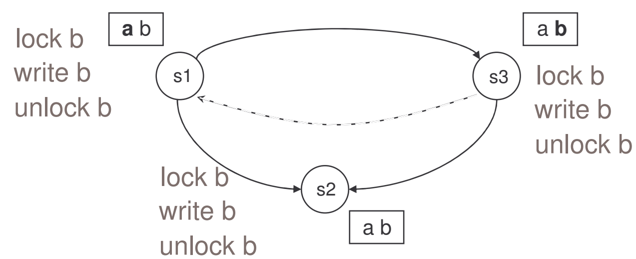


- How many predecessors does s3 have?

Introduction
DAG(WT)
DAG(T)
BackEdge

BackEdge Protocol

1. After execution of T_i, S_1 is sent to s_1 – no commit, locks are not released.
2. S_2, ..., S_j – as before, but without commit and without releasing locks.
3. 2PC for T_i, S_1, ..., S_j.
4. Remaining secondary subtransactions: lazy, as in DAG(T).



Introduction
DAG(WT)
DAG(T)
BackEdge

BackEdge Protocol – Discussion

- ‚No magic‘ here.
- Better than primary-copy schemes, according to simulations. Upto Factor 5, if there are many readers and few backedges.
- Implementation on top of commercial DBMSs relatively easy.

Literature

- Yuri Breitbart et al.: Update Propagation Protocols for Replicated Databases. Proceedings SIGMOD'99.

Potential Exam Questions

- Illustrate that lazy replication schemes, when designed carelessly, may lead to inconsistencies.
- Why is the topology of the copy graph important in the context of lazy replication?
- Explain the different approaches from the lecture that ensure consistency with lazy replication.