Chapter 7: Lazy Replication

Update Propagation Protocols (1)

1. Alternatives to update replicas: synchronous vs. asynchronous:
   - synchronous – within transaction (eager replication),
   - asynchronous – separate transaction to update replicas (lazy replication).

1. 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)

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

Update Propagation Protocols (3)

1. 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.
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: \(w_2[b]\) \(r_3[a]\) \(r_3[b]\) \(w_1[a]\)

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.

System Model/Assumptions

1. Each data object has a primary site.
   - primary copy,
   - secondary copies/replicas.
2. Transaction has an originating site.
3. Transaction can only modify data objects whose primary site = originating site; but may read any data object.
4. Nodes (sites) use 2PL.
5. Network is reliable; delivery of messages in FIFO order.
6. Primary subtransaction, secondary subtransaction.
Copy Graph

Copy Graph:
1. nodes = sites.
2. Edge from s_i to s_j iff primary copy of a data object is on s_i, and ∃ secondary copy on s_j.
3. 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)

1. "DAG without Timestamps"
2. Prerequisite: copy graph is cycle-free.
3. Generate Tree T from copy graph: s_i child of s_j ⇒ s_i successor of s_j in T.
4. Example:

DAG(WT) Protocol (2)

1. Node forwards update transactions to children in T.
2. Commit order
   - order in which transactions arrive at node
   - order in which transactions are forwarded to children.
**DAG(WT) Protocol (3)**

1. 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)**

1. Point that is still open (cont.):
   - Local deadlocks – illustration:
     - T_1: r_1(c) w_1(c), 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**

1. 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.
2. 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 – Observation**

1. Transaction is routed to nodes that are not relevant.
2. Delays.
DAG(T) Protocol

1. "DAG with Timestamps"
2. Propagate update transactions along edges of copy graph.
3. Primary subtransactions have timestamp that specify execution order.
4. Outline of the following:
   - Structure of timestamp (total order necessary),
   - protocol itself.
5. Structure of timestamps is interesting, since their generation is decentralized.

Introduction

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

Timestamps (1)

1. In what follows:
   - auxiliary notion local timestamp counter,
   - auxiliary notion timestamp of a site,
   - timestamp of a transaction.
2. Acyclicity results in total order of sites; s_i < s_{i+k}
3. auxiliary construct –
tuple corresponding to Node s_i : (s_i, LTS_i)
LTS = Local Timestamp Counter;
counts the primary subtransactions that have committed there.

Introduction

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

Local Timestamp Counters – Example

(s1, 0), (s2, 0), (s3, 0)
(s1, 1)
(s2, 1)

 Timestamps (2)

1. 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.
2. Timestamp of a site reflects how many primary subtransactions and how many secondary subtransactions have committed there.
**Example:**

<table>
<thead>
<tr>
<th>Site</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>(s1, 0)</td>
</tr>
<tr>
<td>s2</td>
<td>(s1, 0), (s2, 0)</td>
</tr>
<tr>
<td>s3</td>
<td>(s1, 0), (s2, 0), (s3, 0)</td>
</tr>
</tbody>
</table>

**Example:**

<table>
<thead>
<tr>
<th>Site</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>(s1, 1)</td>
</tr>
<tr>
<td>s2</td>
<td>(s1, 0), (s2, 0)</td>
</tr>
</tbody>
</table>

**Example:**

<table>
<thead>
<tr>
<th>Site</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>(s1, 1)</td>
</tr>
<tr>
<td>s2</td>
<td>(s1, 0), (s2, 1)</td>
</tr>
</tbody>
</table>

**Example:**

<table>
<thead>
<tr>
<th>Site</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>(s1, 1)</td>
</tr>
<tr>
<td>s2</td>
<td>(s1, 0), (s2, 1)</td>
</tr>
</tbody>
</table>

**Example:**

<table>
<thead>
<tr>
<th>Site</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>(s1, 1)</td>
</tr>
<tr>
<td>s2</td>
<td>(s1, 1), (s2, 1)</td>
</tr>
</tbody>
</table>

**Order of timestamps shall reflect commit order.**

Let TS$_1$ < TS$_2$ be a lexicographic ordering of timestamps.

- TS$_1$ is a prefix of TS$_2$.
- 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$_j$.
**Data Structures**

Data structure for each node:

1. timestamp vector of the node (= timestamp of the last committed secondary subtransaction + tuple of the node),
2. 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)

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

Secondary Transaction (2)

1. After commit: $TS(s_i) := TS(T_i)(s_i, LTS_i)$
2. 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

1. $T_1$ has Timestamp $(s_1, 1)$,
2. $T_2$ has Timestamp $(s_1, 1), (s_2, 1)$. 
   (When $T_1$ commits on $s_2$, the site timestamp is set to $(s_1, 1), (s_2, 0)$.)
3. 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

1. Copy graph must be acyclic for DAG(WT) and DAG(T) protocols.
2. 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**

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

---

**BackEdge Protocol – Terminology (1)**

1. Backedge from $s_i$ to $s_j$.
2. $s_j$ is predecessor of $s_i$ in $G_{\text{dag}}$.
3. $T_i$ – primary subtransaction with node $s_i$.
4. „Backedge subtransactions“: transactions $S_{i1}$, ..., $S_{ij}$ at predecessor $s_{i1}$, ..., $s_{ij}$ of $s_i$ in $T$.
5. $s_{i1}$ “most distant” of $s_i$ etc.

---

**BackEdge Protocol – Terminology (2)**

1. Illustration:
2. How many predecessors does $s_3$ have?

---

**BackEdge Protocol**

1. After execution of $T_i$, $S_i$ is sent to $s_i$ – no commit, locks are not released.
2. $S_2$, ..., $S_j$ – as before, but without commit and without releasing locks.
3. 2PC for $T_i$, $S_{i1}$, ..., $S_{ij}$.
4. Remaining secondary subtransactions: lazy, as in DAG(T).
**BackEdge Protocol – Discussion**

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

**Potential Exam Questions**

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

**Literature**