Timestamp Ordering

Assign each transaction a unique *timestamp (ts)*

- Serialize transactions according to timestamps

Keep track of timestamp last transaction to read and write an object

**Invariants**

1. If T reads O, last write timestamp must be lower than T
2. If T writes O, last read and write timestamp must be lower than T’s

If T tries to read/write object with higher timestamp, abort and rollback

T (1)  U (2)  V (3)
read X (X.rts=1)  write Y(Y.wts=1)
read X (X.rts=2)  read Y (Y.rts = 3)  write X (X.wts=3)
read Y (Y.rts=3)  write X: abort!
Timestamp Ordering Invariants

Let T1 and T2 have timestamps 1 and 2

Invariants enforce order T1 ; T2

I1: If T reads O, last write timestamp must be lower than T

• If T1 reads O after T2 writes O, T1 sees T2’s write
Timestamp Ordering Invariants

Let T1 and T2 have timestamps 1 and 2

Invariants enforce order T1 ; T2

I1: If T reads O, last write timestamp must be lower than T
   • If T1 reads O after T2 writes O, T1 sees T2’s write

I2: If T writes O, last read and write timestamp must be lower than T’s
   • If T2 reads O before T1 writes O, T2 missed T1’s write

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read O</td>
</tr>
<tr>
<td>write O</td>
<td></td>
</tr>
</tbody>
</table>
Timestamp Ordering Invariants

Let T1 and T2 have timestamps 1 and 2
Invariants enforce order T1 ; T2

I1: If T reads O, last write timestamp must be lower than T
• If T1 reads O after T2 writes O, T1 sees T2’s write

I2: If T writes O, last read and write timestamp must be lower than T’s
• If T2 reads O before T1 writes O, T2 missed T1’s write
• If T1 writes O after T2 writes O, T2’s write has been lost
Thomas Write Rule

Let $T_1$ and $T_2$ have timestamps 1 and 2

Invariants enforce order $T_1; T_2$

$I_1$: If $T$ reads $O$, last write timestamp must be lower than $T$

- If $T_1$ reads $O$ after $T_2$ writes $O$, $T_1$ sees $T_2$’s write

$I_2$: If $T$ writes $O$, last read and write timestamp must be lower than $T$’s

- If $T_2$ reads $O$ before $T_1$ writes $O$, $T_2$ missed $T_1$’s write
- If $T_1$ writes $O$ after $T_2$ writes $O$, $T_2$’s write has been lost

If $T$ writes $O$ and last write timestamp > $T$’s, skip write!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>write $O$</td>
</tr>
<tr>
<td>write $O$</td>
<td></td>
</tr>
</tbody>
</table>
Should we abort or skip here?

T (1) U (2) V (3)
read X (X.rts=1)
write Y (Y.wts=1)
  read X (X.rts=2)
   read Y (Y.rts = 3)
    write X (X.wts=3)
     read Y (Y.rts=3)
write X: ???
Dependency Tracking

Start with X=0, Y=0, Z=0

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X -&gt; 0</td>
<td></td>
</tr>
<tr>
<td>Write Y = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read Y -&gt; 1</td>
</tr>
<tr>
<td></td>
<td>Write Z = 2</td>
</tr>
<tr>
<td>Read Z—ABORT!</td>
<td></td>
</tr>
</tbody>
</table>
Dependency Tracking

Start with X=0, Y=0, Z=0

T2 has read value that was produced by aborted transaction!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X → 0</td>
<td></td>
</tr>
<tr>
<td>Write Y = 1</td>
<td>Read Y → 1</td>
</tr>
<tr>
<td></td>
<td>Write Z = 2</td>
</tr>
<tr>
<td></td>
<td>Read Z—ABORT!</td>
</tr>
</tbody>
</table>
Dependency Tracking

Start with $X=0$, $Y=0$, $Z=0$

T2 has read value that was produced by aborted transaction!

When reading object $O$, add its RTS to dependency list

At commit time, check dependency list
  • If tx in dependency list has aborted, abort
  • If tx in dependency list is still active, wait

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X $\rightarrow$ 0</td>
<td></td>
</tr>
<tr>
<td>Write Y $\rightarrow$ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read Y $\rightarrow$ 1</td>
</tr>
<tr>
<td></td>
<td>Write Z $= 2$</td>
</tr>
<tr>
<td></td>
<td>Read Z $\rightarrow$ ABORT!</td>
</tr>
</tbody>
</table>
Timestamp Ordering

T (1)    U (2)    V (3)
read X (X.rts=1)
write Y (Y.wts=1)
read X (X.rts=2)
  read Y (Y.rts = 3)
  write X (X.wts=3)
  write Z (Z.wts=3)
read Y (Y.rts=3)
write Z: skip!

T (1)    U(2)    V(3)
read X
write Y
write Z
read X
read Y
write Y
write X
write Z
Timestamp Summary

read(X)
  if WTS(X) > myTS:
    abort()
  myDEPS.add(WTS(X))
  RTS(X) = max(RTS(X), myTS)

write(X)
  if RTS(X) > myTS:
    abort()
  if WTS(X) > myTS:
    return # skip write
  WTS(X) = myTS

At commit time, wait for myDEPS to complete, abort if any has aborted
Distributed Transactions

CS425/ECE428 – Distributed Systems – Spring 2020

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya
Client-Server Transactions

- **Atomicity**: all-or-nothing
  - Make updates on a shadow copy
  - Real update on commit, discard on abort

- **Consistency**: invariants satisfied
  - Check and abort on violations

- **Isolation**: concurrent transactions serially equivalent
  - Two-phase locking (strict or otherwise)

- **Durability**: results preserved after crashes
  - Save committed updates to disk, recover state after crash
Distributed Transactions

• A transaction that invokes operations at several servers.

Flat Distributed Transaction

Nested Distributed Transaction
Coordination in Distributed Transactions

Coordinator & Participants

The Coordination Process

Coordinator

Participant

Open Transaction

TID

Close Transaction

Abort Transaction

a.method (TID)

Join (TID, ref)

Coordinator

Participant

Join (TID, ref)

Coordinator

Participant

Join (TID, ref)
Distributed banking transaction

\[ T = \text{openTransaction} \]
\[ a.\text{withdraw}(4); \]
\[ c.\text{deposit}(4); \]
\[ b.\text{withdraw}(3); \]
\[ \text{closeTransaction} \]

Note: the coordinator is in one of the servers, e.g. BranchX
Distributed Transaction Challenges

• **Atomicity**: all-or-nothing
  • Must ensure atomicity across servers

• **Consistency**: invariants satisfied
  • Generally done locally, but may need to check non-local invariants at commit time

• **Isolation**: concurrent transactions serially equivalent
  • Locks at each server.

• **Durability**: results preserved after crashes
  • Each server keeps local recovery log
I. Locks in Distributed Transactions

- Each server is responsible for applying concurrency control to objects it stores.
- Servers are collectively responsible for serial equivalence of operations.
- Locks are held locally and cannot be released until all servers involved in a transaction have committed or aborted.
- Locks are retained during 2PC protocol.
- Since lock managers work independently, deadlocks are possible (likely?)
Distributed Deadlocks

• The wait-for graph in a distributed set of transactions is distributed

• Centralized detection
  • Each server reports waits-for relationships to coordinator
  • Coordinator constructs global graph, checks for cycles

• Decentralized — edge chasing
  • Forward “probe” messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.
Probes Transmitted to Detect Deadlock

$W \rightarrow U \rightarrow V \rightarrow W$

Deadlock detected

Held by

Waits for

$W \rightarrow U \rightarrow V$

Initiation

$W \rightarrow U$

Waits for

$W \rightarrow U$

Held by

Waits for
Edge Chasing

• **Initiation**: When a server $S_1$ notices that a transaction $T$ starts waiting for another transaction $U$, where $U$ is waiting to access an object at another server $S_2$, it initiates detection by sending $<T \to U>$ to $S_2$.

• **Detection**: Servers receive probes and decide whether deadlock has occurred and whether to forward the probes.

• **Resolution**: When a cycle is detected, one or more transactions in the cycle is/are aborted to break the deadlock.

• Phantom deadlocks=false detection of deadlocks that don’t actually exist
  
  • Edge chasing messages contain stale data (Edges may have disappeared in the meantime). So, all edges in a “detected” cycle may not have been present in the system all at the same time.
Transaction Priority

• Transactions are given priorities
  • E.g., inverse of timestamp
  • Total order

• When deadlock cycle is found, abort lowest priority transaction
  • Only one aborted even if several simultaneous probes find cycle
II. Atomic Commit Problem

- At some point, client executes closeTransaction()
  - Result -> commit, abort
- Atomicity requires all-or-nothing
  - All operations on all servers are committed, or
  - All operations on all servers are aborted
- What problem statement is this?
Atomic Commit Protocols

- **Consensus!**
  - Impossible to be totally correct
  - Possible to ensure safety, at the (possible) expense of liveness
  - Plus, we already have a leader (coordinator)

- **First attempt: Coordinator decides**
  - Pick commit or abort
  - Send message to all participants
  - (Retransmit until acknowledged)

- **Problems?**
  - Participant crashes before receiving commit message
  - Participant decides to abort (deadlock, other problems)