Outline

• Today:
  – Replication consistency
  – Frameworks that support consistency
    • Zookeeper (Apache project)
Transactions and deadlocks

Part 1
Deadlocks

• Deadlock occurs when a loop is created in a logical \textit{waits for} graph
  – Transaction T acquires A, Transaction U acquires B
  – Transaction T waits for B, Transaction U waits for A
  – Neither transaction able to make forward progress

• Loop can be arbitrarily long
  – $T \rightarrow U \rightarrow V \rightarrow W \rightarrow \ldots \rightarrow Z \rightarrow T$
  – Typically, loops are short (one hop)
## Deadlock Example

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Withdraw(A, 4); Deposit(B, 4)</strong></td>
<td><strong>Withdraw(C, 3); Deposit(B, 3)</strong></td>
</tr>
<tr>
<td><strong>Balance = A.Read()</strong></td>
<td><strong>Balance = C.Read()</strong></td>
</tr>
<tr>
<td><strong>A.Write(balance-4)</strong></td>
<td><strong>C.Write(balance-3)</strong></td>
</tr>
<tr>
<td><strong>Balance = B.read()</strong></td>
<td><strong>Balance = B.Read()</strong></td>
</tr>
</tbody>
</table>
# Deadlock Example

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
</tr>
<tr>
<td>Balance=A.Read()</td>
<td>Balance = C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance=B.read()</td>
<td>Balance=B.Read()</td>
</tr>
<tr>
<td>B.Write(balance+4)</td>
<td>Wait U</td>
</tr>
<tr>
<td>Read A</td>
<td>Read B</td>
</tr>
<tr>
<td>Read A</td>
<td>Read B</td>
</tr>
</tbody>
</table>

**Time**
# Deadlock Example

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdraw(A, 4); Deposit(B, 4)</td>
<td>Withdraw(C, 3); Deposit(B, 3)</td>
</tr>
<tr>
<td>Balance=A.Read()</td>
<td>Balance = C.Read()</td>
</tr>
<tr>
<td>A.Write(balance-4)</td>
<td>C.Write(balance-3)</td>
</tr>
<tr>
<td>Balance=B.read()</td>
<td>Balance=B.Read()</td>
</tr>
<tr>
<td>B.Write(balance+4)</td>
<td>B.Write(balance+3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Deadlock
Waits-For Graph

• Cycle: T → B → U → B → T
Deadlock Prevention

• Gather *all* locks at beginning of transaction
  – Can we still run into problems?
# Deadlock Prevention

- Gather *all* locks at beginning of transaction
  - Can we still run into problems? Yes, if locks are acquired in different order

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Transaction U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock A</td>
<td>Lock B</td>
</tr>
<tr>
<td>Lock B (Wait)</td>
<td>Lock A (Wait)</td>
</tr>
</tbody>
</table>

*Deadlock*
Deadlock Prevention

• Gather *all* locks at beginning of transaction
  – Can we still run into problems? Yes, if locks are acquired in different order
  – Solution: gather locks in canonical order
• Won’t work in many circumstances
  – Cannot predict locking requirements of interactive apps
  – Unnecessarily reduces concurrency
    • E.g., acquire read (write) lock for long-running transaction
    • Prevent *any* other transaction from acquiring write (read) lock
Deadlock Detection

• Concurrent thread keeps track of waits-for graph
  – Typically lock manager
  – For each successful lock operation, track resources held by each transaction
  – For each conflicting lock operation (condition var wait), track transaction waits for relationship
  – On lock release, delete edges corresponding to signaled transactions
Deadlock Detection

• Search for loops in waits for graph on adding edge
  – Abort one transaction in the loop
  – Release all locks associated with transaction
    • Releasing locks signals blocked transaction in loop, break deadlock
  – Aborted transaction must restart
    • Starvation

• Which transaction to abort?
Deadlock Detection

• Search for loops in waits for graph on adding edge
  – Abort one transaction in the loop
  – Release all locks associated with transaction
    • Releasing locks signals blocked transaction in loop, break deadlock
  – Aborted transaction must restart
    • Starvation

• Which transaction to abort?
  – Assign priorities to transactions?
  – Shortest running transaction?
Preventing Deadlocks Through Timeouts

• Assign a timeout with each lock
  – When timeout expires, lock becomes vulnerable

• If no other transaction waiting for lock, continue as normal

• If other transaction waiting, abort transaction holding vulnerable lock

• Pros/cons?
Preventing Deadlocks Through Timeouts

• Assign a timeout with each lock
  – When timeout expires, lock becomes vulnerable
• If no other transaction waiting for lock, continue as normal
• If other transaction waiting, abort transaction holding vulnerable lock

• Pros/cons
  – Pros: simpler to implement?
  – Cons: abort transaction when no deadlock exists, what value to set for timeout (lightly loaded vs. heavily loaded system)
Distributed Transactions

Part 2
Distributed Transactions

• Client makes atomic request that accesses resources at multiple databases
  – Potentially spread across wide area
• Distributed transaction can be:
  – *Simple*: Client explicitly accesses resources at multiple sites
  – *Nested*: transaction spawns one-or more sub-transactions
Simple Distributed Transaction

- Client
- X
- Y
- Z
Nested Distributed Transaction

Client

T

X

Y

Z
Distributed Transaction

• In distributed transactions, some server must be responsible for either committing or aborting transaction at all sites

• Typically, first server contacted becomes *coordinator*
  – Client tracks identity of coordinator
  – Tells each additional server (*workers*) of coordinator identity
  – Workers responsible for registering with coordinator

• Transaction abort/commit through coordinator
  – Client commit request relayed to workers
  – Worker abort request relayed to other workers
Distributed Transaction Commit Protocol

• One option: client communicates to coordinator desire to abort or commit
  – Coordinator communicates decision to all workers
  – One-phase protocol
  – Problems?
Distributed Transaction Commit Protocol

• One option: client communicates to *coordinator* desire to abort or commit
  – Coordinator communicates decision to all workers
  – One-phase protocol
  – Does not allow individual servers to voice their opinion
    • Failure, concurrency control (deadlock)
Two-Phase Commit

• Phase one (voting)
  – Coordinator sends *CanCommit*? request to each worker
  – Each worker replies with vote (yes/no)

• Phase two (complete based on vote)
  – If no failures and everyone votes yes, coordinator sends *DoCommit* to each worker
  – Otherwise, send *AbortTransaction* to each worker
  – On success, each worker sends *HaveCommitted* to coordinator
Two-Phase Commit

1. CanCommit?
2. Yes
3. DoCommit
4. HaveCommit

1. Coord: Prepared to Commit
2. Worker: Prepared to Commit (uncertain)
3. Coord: Committed
4. Worker: Committed (done)
Apache Zookeeper:
Software support for distributed transactions and consistency

Part 3
Zookeeper Background

Common needs
• Keep configuration files across cluster consistent
• Leader election
  – Which server handles updates?
  – Even when there are failures?
• Distributed locks and transactions

Idea: a “file system” like API
• Provide applications with a pseudo-filesystem API
• Create, delete, and update objects in hierarchy
• Properties:
  – FIFO client ordering
  – Linearizable semantics
    • Via underlying “Zab” protocol
• Clients get ‘updates’
  – Can “watch” for changes in the file system hierarchy
Zookeeper data tree

- Hierarchical namespace
  - Called a data tree
- Tree of ZNodes
  - Like files in a file system
  - Can hold arbitrary key-value data
- This “file system” is designed for metadata
  - Not the data itself
  - E.g., the blocklist, not the block contents for Project 2
Client interface to Zookeeper

• Create(path, data, ...)
  – “regular” or sequential names
• Delete(path, v)
  – Del if path exists at version ‘v’
• Exists(path, watch)
  – Path exists? Can also set watch to be notified if it exists in the future
• getData(path, watch); setData(path, data)
• getChildren(path, watch)
Ensuring ZooKeeper guarantees

- All servers store a copy of the data (in memory)
- A leader is elected at startup
- “Follower” servers service clients, but all updates go through leader
- Update responses are sent when a majority of servers have persisted the change
  - Quorums in action!
- ZAB protocol ensures consistency via Atomic Broadcast and single leader election
  - Since leader sees the update, can turn into idempotent update and reliably broadcast to the quorum
Zookeeper example

• Recall that ZooKeeper allows clients to “watch” for changes in the namespace
  – New directories/files
  – Deleted directories/files
  – New children of a ZNode
  – Etc.
• Any updates to the “filesystem” are sent through the master, and don’t commit until a majority of the quorum records the change
• Master is single point of serialization
  – All clients see the same state
• Master can fail over if it fails
  – Fault tolerance
Implementing locks

**Lock**

```
1 n = create(l + "\lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

**Unlock**

```
1 delete(n)
```