Replication, fault tolerance, and load balancing

George Porter
CSE 124
Mar 3, 2015
Announcements

- Project 3
  - Concurrency support: two kinds
    1. Internal to your server (due to multithreading)
    2. Between servers (replication, consensus, etc)
  - For your project, just focus on the first case
  - OK to use “global lock”
  - Due on Thursday March 12th
public class MyClass {
    private Object mutex;
    public MyClass() {
        mutex = new Object();
    }

    public void myMethod() {
        synchronized(mutex) {
            ...
        }
    }
}
Alternatively

As long as each method “stands on its own”

Don’t leak internal state to other objects/threads

```java
class SynchronizedCounter {
    private int c = 0;
	public synchronized void increment() {
	    c++;
    }

    public synchronized void decrement() {
	    c--;
    }

    public synchronized int value() {
	    return c;
    }
}
```
Transactions
ACID Property

- **Atomicity**
  - All or nothing (even in the face of failure)

- **Consistency**
  - Take system from one consistent state to another

- **Isolation**
  - No interference from concurrent transactions: do not see the intermediate (uncommitted) results of any transaction

- **Durability**
  - Once *committed* all effects are saved in permanent storage
Serial Equivalence

- One approach to avoiding concurrency problems is to execute transactions in *serial* order
  - Do not begin transaction \( n+1 \) until transaction \( n \) either commits or aborts
  - Results in unacceptable performance
  - What if transactions are not accessing the same data items?

- Another approach is to ensure that transactions execute in *serial equivalent* order
  - Interleave operations only when it is safe to do so
  - Final result same as if transactions performed in serial order

- How to achieve serial equivalence?
So …

■ Need to provide *protocol* (rules on how data item is accessed) to ensure conflict serializability

■ Goal of protocol:
  • To allow access for data items that are not required by multiple transactions
  • For those data items required by multiple transaction, restrict access in some way, or limit it to exclusive access

■ Balance between safety and efficiency
  • Too restrictive: little or no concurrency, ineffective
  • Too lenient: leads to inconsistency
Lock-based protocols

- “Exclusive” access $\Rightarrow$ locks
- Each database item is associated with locks
- Transaction must obtain locks before accessing the object
- Transaction must release lock when it finishes.
**Lock-based protocols**

- **Example:**

  \[
  \text{Read}(X) \quad \Rightarrow \quad \text{Lock}(X) \quad \Rightarrow \quad \text{Read}(X) \quad \Rightarrow \quad \text{Unlock}(X)
  \]

- **Lock**(X): check if object X is already locked
  - If not, obtain the lock
  - If so, wait or “do something” to handle the potential deadlock (like aborting)

- One does not have to read immediately after locking

- **Unlock**(X): release the lock on object X

- The addition of **Lock**(X) and **Unlock**(X) commands are done by the DBMS
Lock-based protocols: S and X locks

- How many transactions can obtain the lock to an item?
  - One
- Too restrictive?
- Consider the two transactions

  1. Read(X)         1. Read(Y)
  2. Read(Y)         2. Read(X)
  3. Read(Z)         3. Read(Z)

  T1                  T2

- There seems to be no reason for one transaction to wait for the other
Lock-based protocols: S and X locks

- Two kinds of locks on each object
  - Shared locks (S-locks)
    - Requested before reading
    - Multiple transactions can hold a S-lock on an object simultaneously
  - Exclusive locks (X-locks)
    - Requested before writing
    - Only one transaction can hold an X-lock on an object at any given time
    - No other transaction can hold any lock (not even a S-lock) if some transaction has an X-lock
## Lock-based protocols: S and X locks

- More on S and X locks
  - A transaction that holds an S-lock on an object can read the object
  - A transaction that holds an X-lock on an object can read and write the objects

- Lock-compatibility table

<table>
<thead>
<tr>
<th>T1 Request</th>
<th>S-lock</th>
<th>X-lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-lock</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>X-lock</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
Achieving Serial Equivalence

- Approaches to serial equivalence
  - Locking
  - Optimistic concurrency control (check on commit)

- Achieve serial equivalence by locking necessary data items
  - Avoiding deadlock
  - Breaking deadlock
  - Granularity of locks
  - Read/write (shared/exclusive) locks
  - Two phase commit for distributed lock management
Two-Phase Locking

- Serial equivalence requires that all data accesses be serialized w/respect to other transactions
  - If two transactions conflict, then *all* operations within transactions must be (logically) done either before or after other transaction

- Two-phase locking: do not acquire any new locks after releasing any lock
  - Growing phase, followed by shrinking phase

- Strict two-phase locking
  - Must worry about aborted transactions (dirty writes)
  - Do not release locks until transaction aborts/commits
Lock Manager

- Want to automate the process of acquiring/releasing locks
  - Avoid forcing programmers to get it right every time
  - Use API to force all data accesses go through lock manager

- Inside transaction, on data access:
  - If object not locked, server locks it
  - If conflicting lock, block transaction (Wait on condition var)
  - If non-conflicting lock, acquire shared lock
  - For read/write access consider lock promotion

- On transaction commit/abort:
  - Release all locks, signal appropriate waiting transactions
  - Starvation?
Deadlock: getting “stuck”
Deadlocks

- Deadlock occurs when a loop is created in a logical 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>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>
</tbody>
</table>

**time**

- Read A
- Read C
- Read B
- Read B
**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></td>
</tr>
<tr>
<td>Wait U</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**
- **time**
- **Transaction T**
  - Withdraw(A, 4)
  - Deposit(B, 4)
  - Read A
  - Write(balance-4)
  - Wait U
- **Transaction U**
  - Withdraw(C, 3)
  - Deposit(B, 3)
  - Read C
  - Write(balance-3)
  - Wait U
## 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>
</tbody>
</table>

```
A.Write(balance-4)
B.Write(balance+4)
```

```
C.Write(balance-3)
B.Write(balance+3)
```

---

**Deadlock**
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 A (Wait)</td>
</tr>
<tr>
<td>Lock B (Wait)</td>
<td>Lock B</td>
</tr>
<tr>
<td></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*
    - Not strictly necessary
  - 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 \textit{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
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
Nested Distributed Transaction
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

Next 6 slides thanks to Mahadev Konar
What is ZooKeeper?

A highly available, scalable, distributed coordination kernel
Use Cases

» Leader Election
» Group Membership
» Work Queues
» Event Notifications/workflow management
» Configuration Management
» Cluster Management
» Sharding
What is ZooKeeper again?

- File api without partial reads/writes
- No renames
- Ordered updates and strong persistence guarantees
- Conditional updates (version)
- Watches for data changes
- Ephemeral znodes
- Generated file names
Data Model

- Hierarchical namespace
- Each znode has data and children
- Data is read and written in its entirety
ZooKeeper API

String create(path, data, acl, flags)

void delete(path, expectedVersion)

Stat setData(path, data, expectedVersion)

(data, Stat) getData(path, watch)

Stat exists(path, watch)

String[] getChildren(path, watch)
ZooKeeper Service

- All servers store a copy of the data (in memory)
- A leader is elected at startup
- Followers service clients, all updates go through leader
- Update responses are sent when a majority of servers have persisted the change