Principles of Software Construction: Objects, Design and Concurrency

Data consistency

Jonathan Aldrich  Charlie Garrod

© 2012 C Garrod, J Aldrich, and W Scherlis
Administrivia

- Homework 9 due Thursday
  - Commit to team repository if working as team
  - Also don't forget to turn in Lab 09
Last time: Distributed systems

- Caching and partitioning for scalability
  - Consistent hashing
  - Master/tablet-based systems
Today: Data consistency and concurrency control

- A formal definition of consistency
- Introduction to transactions
- Introduction to concurrency control
- Distributed concurrency control
  - Two-phase commit
An aside: Double-entry bookkeeping

• A style of accounting where every event consists of two separate entries: a credit and a debit

```java
void transfer(Account fromAcct, Account toAcct, int val) {
    fromAccount.debit(val);
    toAccount.credit(val);
}

static final Account BANK_LIABILITIES = …;

void deposit(Account toAcct, int val) {
    transfer(BANK_LIABILITIES, toAcct, val);
}

boolean withdraw(Account fromAcct, int val) {
    if (fromAcct.getBalance() < val) return false;
    transfer(fromAcct, BANK_LIABILITIES, val);
    return true;
}
```
Some properties of double-entry bookkeeping

- Redundancy!
- Sum of all accounts is static
  - Can be 0
Data consistency of an application

• Suppose $\mathcal{D}$ is the database for some application and $\varphi$ is a function from database states to $\{\text{true, false}\}$
  ▪ We call $\varphi$ an *integrity constraint* for the application if $\varphi(\mathcal{D})$ is true if the state $\mathcal{D}$ is "good"
  ▪ We say a database state $\mathcal{D}$ is *consistent* if $\varphi(\mathcal{D})$ is true for all integrity constraints $\varphi$
  ▪ We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint $\varphi$
Data consistency of an application

- Suppose $\mathcal{D}$ is the database for some application and $\varphi$ is a function from database states to $\{\text{true, false}\}$
  - We call $\varphi$ an integrity constraint for the application if $\varphi(\mathcal{D})$ is true if the state $\mathcal{D}$ is "good"
  - We say a database state $\mathcal{D}$ is consistent if $\varphi(\mathcal{D})$ is true for all integrity constraints $\varphi$
  - We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint $\varphi$

- E.g., for a bank using double-entry bookkeeping one possible integrity constraint is:

```python
def IsConsistent(D):
    if ( sum(all account balances in D) == 0 ):
        return True
    else:
        return False
```
Database transactions

- A *transaction* is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another
  - "Atomic" ~ indivisible

- Transactions always terminate with either:
  - *Commit*: complete transaction's changes successfully
  - *Abort*: undo any partial work of the transaction
A transaction is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another

- "Atomic" ~ indivisible

Transactions always terminate with either:

- Commit: complete transaction's changes successfully
- Abort: undo any partial work of the transaction

```java
boolean withdraw(Account fromAcct, int val) {
    begin_transaction();
    if (fromAcct.getBalance() < val) {
        abort_transaction();
        return false;
    }
    transfer(fromAcct, BANK_LIABILITIES, val);
    commit_transaction();
    return true;
}
```
A functional view of transactions

- A transaction $T$ is a function that takes the database from one state $D$ to another state $T(D)$
- In a correct application, if $D$ is consistent then $T(D)$ is consistent for all transactions $T$
A functional view of transactions

• A transaction $T$ is a function that takes the database from one state $D$ to another state $T(D)$

• In a correct application, if $D$ is consistent then $T(D)$ is consistent for all transactions $T$
  - E.g., in a correct application any serial execution of multiple transactions takes the database from one consistent state to another consistent state
Database transactions in practice

- The application requests commit or abort, but the database may arbitrarily abort any transaction
  - Application can restart an aborted transaction

- Transaction ACID properties:
  - Atomicity: All or nothing
  - Consistency: Application-dependent as before
  - Isolation: Each transaction runs as if alone
  - Durability: Database will not abort or undo work of a transaction after it confirms the commit
Concurrent transactions and serializability

- For good performance, database interleaves operations of concurrent transactions
Concurrent transactions and serializability

• For good performance, database interleaves operations of concurrent transactions

• Problems to avoid:
  ▪ Lost updates
    • Another transaction overwrites your update, based on old data
  ▪ Inconsistent retrievals
    • Reading partial writes by another transaction
    • Reading writes by another transaction that subsequently aborts

• A schedule of transaction operations is *serializable* if it is equivalent to some serial ordering of the transactions
  ▪ a.k.a. *linearizable*
Concurrency control for a centralized database

- **Two-phase locking (2PL)**
  - Phase 1: acquire locks
  - Phase 2: release locks

- **E.g.,**
  - Lock an object before reading or writing it
  - Don't release any locks until commit or abort
Concurrency control for a distributed database

- **Distributed two-phase locking**
  - Phase 1: acquire locks
  - Phase 2: release locks

- **E.g.,**
  - Lock all copies of an object before reading or writing it
  - Don't release any locks until commit or abort

- **Two new problems:**
  - Distributed deadlocks are possible
  - All participants must agree on whether each transaction commits or aborts
Two-phase commit (2PC)

- **Two roles:**
  - Coordinator: for each transaction there is a unique server coordinating the 2PC protocol
  - Participants: any server storing data locked by the transaction

- **Two phases:**
  - Phase 1: Voting (or Prepare) phase
  - Phase 2: Commit phase

- **Failure model:**
  - Unreliable network:
    - Messages may be delayed or lost
  - Unreliable servers with reliable storage:
    - Servers may crash or temporarily fail
    - Will eventually recover persistently-stored state
The 2PC voting phase

- Coordinator sends `canCommit?()` message to each participant
  - Messages re-sent as needed

- Each participant replies `yes` or `no`
  - May not change vote after voting
    - Must log vote to persistent storage
    - If vote is `yes`:
      - Objects must be strictly locked to prevent new conflicts
      - Must log any information needed to successfully commit

- Coordinator collects replies from participants
The 2PC commit phase

• **If participants unanimously voted yes**
  - Coordinator logs `commit(T)` message to persistent storage
  - Coordinator sends `doCommit(T)` message to all participants
    • Participants confirm, messages re-sent as needed

• **If any participant votes no**
  - Coordinator sends `doAbort(T)` message to all participants
    • Participants confirm, messages re-sent as needed
Coordinator:

“prepared”

\[ \text{canCommit?} \]

yes

“committed” (persistently)

\[ \text{doCommit} \]

confirmed

“done”

Participants:

“prepared” (persistently)

“uncertain” (objects still locked)

“committed”
Problems with two-phase commit?
Problems with two-phase commit?

• Failure assumptions are too strong
  ▪ Real servers can fail permanently
  ▪ Persistent storage can fail permanently

• Temporary failures can arbitrarily delay a commit

• Poor performance
  ▪ Many round-trip messages
The CAP theorem for distributed systems

• For any distributed system you want...
  - Consistency
  - Availability
  - tolerance of network Partitions

• ...but you can support at most two of the three
Next time…

- Ghost of Objects Present
- Ghost of Objects Past
- Ghost of Objects Yet to Come