Concurrency Control

Transactions and Distributed Transactions

Announcements

- 1) Exam Thursday
- 2) Exam review session
- 3) Homework 1 back
- 4) dga,vrv out until Thursday

Last time: RAID

- Trade capacity for reliability
- Throughput growing…delay not

Today: Concurrency control

- Local concurrency control
  - Transactions
  - Two-phase locking
- Distributed concurrency control
  - Two-phase commit

Transactions

- Fundamental abstraction to group operations into a single unit of work
  - **`begin`**: begins the transaction
  - **`commit`**: attempts to complete the transaction
  - **`rollback`** / **`abort`**: aborts the transaction

ACID properties

- Atomicity: all or nothing
- Consistency: guarantee basic properties
- Isolation: each transaction runs as if alone
- Durability: cannot be undone

The classic debit/credit example

```
bool xfer(Account src, Account dest, long x) {
    if (src.getBalance() >= x) {
        src.setBalance(src.getBalance() - x);
        dest.setBalance(dest.getBalance() + x);
        return TRUE;
    }
    return FALSE;
}
```

- If not isolated and atomic:
  - might overdraw the src account
  - might “create” or “destroy” money

The classic debit/credit example

```
bool xfer(Account src, Account dest, long x) {
    Transaction t = begin();
    if (src.getBalance() >= x) {
        src.setBalance(src.getBalance() - x);
        dest.setBalance(dest.getBalance() + x);
        return t.commit();
    }
    t.abort();
    return FALSE;
}
```

- Note: the system is allowed to unilaterally abort the transaction itself, when you try to commit!

Problems to avoid

- Lost updates
  - Another transaction overwrites your change based on a previous value of some data
- Inconsistent retrievals
  - You read data that can never occur in a consistent state
    - partial writes by other transactions
    - writes by a transaction that later aborts

A poor solution: a global lock

- Only let one transaction run at a time
  - isolated from all other transactions
  - make changes permanent on commit or undo changes on abort, if necessary

```
bool xfer(Account src, Account dest, long x) {
    lock();
    if (src.getBalance() >= x) {
        src.setBalance(src.getBalance() - x);
        dest.setBalance(dest.getBalance() + x);
        unlock();
        return TRUE;
    }
    unlock();
    return FALSE;
}
```

Better: lock objects independently

- E.g., one lock for the src account, one lock for the dest account
  - Other transactions can execute concurrently, as long as they don't read or write the src or dest accounts
  - Easy to implement with the tools we have
    - e.g., can use a hash table of lockable objects -> locks

Locks alone are insufficient

- (You need to use the locks correctly)

```
bool xfer(Account src, Account dest, long x) {
    lock(src);
    if (src.getBalance() >= x) {
        src.setBalance(src.getBalance() - x);
        unlock(src);
        lock(dest);
        dest.setBalance(dest.getBalance() + x);
        unlock(dest);
        return TRUE;
    }
    unlock(src);
    return FALSE;
}
```

Allows other transactions to read `src` before we write `dest` and thus see our partially-written state

2-phase locking (2PL)

- Phase 1: acquire locks
- Phase 2: release locks
  - You may not get any more locks after you release any locks
  - Typically implemented by not allowing explicit **`unlock`** calls
    - Locks automatically released on **`commit`**/**`abort`**

Debit/credit with 2PL

```
bool xfer(Account src, Account dest, long x) {
    Transaction t = begin();
    t.lock(src);
    if (src.getBalance() >= x) {
        src.setBalance(src.getBalance() - x);
        t.lock(dest);
        dest.setBalance(dest.getBalance() + x);
        return t.commit();           // unlocks src and dest
    }
    t.abort();                       // unlocks src
    return FALSE;
}
```

2PL might suffer deadlocks

```
t1.lock(foo);
t1.lock(bar);
```

```
t2.lock(bar);
t2.lock(foo);
```

- **`t1`** might get the lock for **`foo`**, then **`t2`** gets the lock for **`bar`**, then both transactions wait while trying to get the other lock

Preventing deadlock

- Each transaction can get all its locks at once
- Each transaction can get all its locks in a predefined order
  - Both of these strategies are impractical:
    - Transactions often do not know which locks they will need in the future

Detecting deadlock

- Construct a “waits-for” graph
  - Each vertex in the graph represents a transaction
  - T1 → T2 if T1 is waiting for a lock T2 holds
- There is a deadlock iff the waits-for graph contains a cycle

“Ignoring” deadlock

- Automatically abort all long-running transactions
  - Not a bad strategy, if you expect transactions to be short
    - A long-running “short” transaction is probably deadlocked

Distributed transactions

- Data stored at distributed locations
- Failure model:
  - messages might be delayed or lost
  - servers might crash, but can recover saved persistent storage

The *coordinator*

- Begins transaction
  - Assigns unique transaction ID
- Responsible for commit/abort
- Many systems allow any client to be the coordinator for its own transactions

The *participants*

- The servers with the data used in the distributed transaction

Problems with simple commit

- “One-phase commit”
  - Coordinator broadcasts “commit!” to participants until all reply
- What happens if one participant fails?
  - Can the other participants then undo what they have already committed?

Two-phase commit (2PC)

- The commit-step itself is two phases
- Phase 1: Voting
  - Each participant prepares to commit, and votes on whether or not it can commit
- Phase 2: Committing
  - Each participant actually commits or aborts

2PC operations

- **`canCommit?(T)`** -> yes/no
  - Coordinator asks a participant if it can commit
- **`doCommit(T)`**
  - Coordinator tells a participant to actually commit
- **`doAbort(T)`**
  - Coordinator tells a participant to abort
- **`haveCommitted(participant,T)`**
  - Participant tells coordinator it actually committed
- **`getDecision(T)`** -> yes/no
  - Participant can ask coordinator if T should be committed or aborted

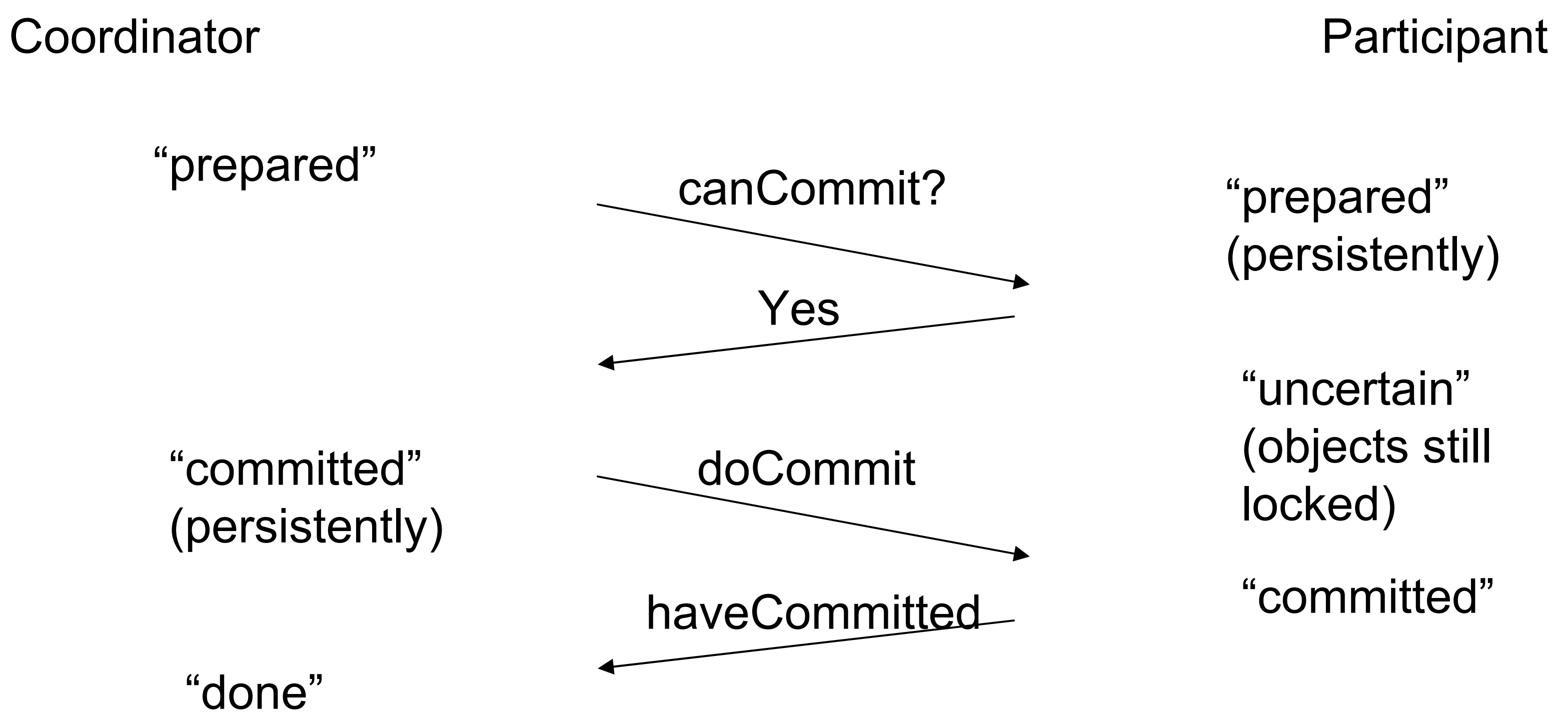
The voting phase

- Coordinator asks each participant: **`canCommit?(T)`**
- Participants must prepare to commit using permanent storage before answering yes
  - Objects are still locked
  - Once a participant votes “yes”, it is not allowed to cause an abort
- Outcome of **`T`** is uncertain until **`doCommit`** or **`doAbort`**
  - Other participants might still cause an abort

The commit phase

- The coordinator collects all votes
  - If unanimous “yes”, causes commit
  - If any participant voted “no”, causes abort
- The fate of the transaction is decided atomically at the coordinator, once all participants vote
  - Coordinator records fate using permanent storage
  - Then broadcasts **`doCommit`** or **`doAbort`** to participants

2PC sequence of events



participant not allowed to cause an abort after it replies “yes” to canCommit

2PL with 2-Phase Commit

- Each participant uses 2PL for its objects, 2PC for the commit process