Notes on Distributed Concurrency Management
15-440, Fall 2012
Carnegie Mellon University
Randal E. Bryant

Reading: Tannenbaum, Sect. 8.5

Part I: Single Server (Not covered very well in book)

Database researchers laid the background for reasoning about how to process a number of concurrent events that update some shared global state. They invented the concept of a "transaction" in which a collection of reads and writes of a global state are bundled such that they appear to be single, indivisible operation. (They also defined standard models for reliable storage and for robust operation. We'll visit those later.)

Desirable characteristics of transaction processing captured by acronym ACID

Atomicity: Each transaction is either completed in its entirety, or aborted. In the latter case, it should have no effect on the global state.

Consistency: Each transaction preserves a set of invariants about the global state. The exact nature of these is system dependent.

Isolation: Each transaction executes as if it were the only one with the ability to read and write the global state.

Durability: Once a transaction has been completed, it cannot be "undone".

(Note that the term "atomicity" or "atomic operations" are often synonomous with the combination of Atomicity+Isolation)

A transaction example.

Imagine we have a set of bank acounts, where balances are stored as an array Bal[i], giving the balance for account i.

We could define a transaction to transfer money from one account to another:

```
xfer(i, j, v):

        if withdraw(i, v):

           deposit(j, v)

        else
           abort
```

where we define operations withdraw & deposit as

```
withdraw(i, v):
    b = Bal[i]                  // Read
    if b >= v                   // Test
        Bal[i] = b-v            // Write
        return true
    else
        return false

deposit(j, v):
    Bal[j] += v
```

Imagine we have Bal[x] = 100, Bal[y] = Bal[z] = 0 and attempt two transactions:

```
T1: xfer(x, y, 60)
T2: xfer(x, z, 70)
```

The ACID properties ensure that any implementation should make it appear as if the two transactions are executed in some serial order:

T1 ; T2. Must have T1 succeed and T2 fail.
End with Bal[x] = 40, Bal[y] = 60.

T2 ; T1. Must have T2 succeed and T1 fail
End with Bal[x] = 30, Bal[y] = 70.

But, without taking special care, we can see that things could go very badly. Consider race condition in updating values of Bal[x]. If transactions interleave between respective Read & Write actions so that Bal[x] ends up as either 30 or 40, but both transactions take place.

This could be viewed as a violation of isolation & durability.

For consistency, consider following function:

```
// Return sum of balances of accounts x & y
sumbalance(i, j, k):
              return Bal[i] + Bal[j] + Bal[k]
```

As a state invariant, we can say sumbalance(x, y, z) == 100 at all times. This got violated due to race, causing money to be artificially created.

Implementing transaction with locks

Easy to wrap lock around entire thing:

```
// Transfer $v from account i to account j
xfer(i, j, v):
        lock()
        if withdraw(i, v):
           deposit(j, v)
        else
           abort
        unlock()
```

But, this would be a serious sequential bottleneck. Prefer to uses finer grained locks, e.g., on a per-account basis:

Attempt #1

```
// Transfer $v from account i to account j
xfer(i, j, v):
        lock(i)
        if withdraw(i, v):
           unlock(i)
           lock(j)
           deposit(j, v)
           unlock(j)
        else
           unlock(i)
           abort
```

There are two problems with this code:

1. Releasing lock i early can give consistency violation. Some other transaction (e.g., sumbalance) could see decremented value of account i, but unincremented value of count j.

Fix: Rule: Only release locks when all updating of state variables completed.

```
xfer(i, j, v):
        lock(i);
        if withdraw(i, v):
           lock(j)
           deposit(j, v)
           unlock(i); unlock(j)
        else
           unlock(i)
           abort
```

2. There's a deadlock. Consider Bal[x] = Bal[y] = 100 and then attempt transactions:

xfer(x, y, 40) and xfer(y, x, 30)

Can reach midpoint both have completed their respective withdrawals, but one holds lock on x while other holds lock on y.

Fixing:

Always acquire locks in a fixed order

```
xfer(i, j, v):
        lock(min(i,j)); lock(max(i,j))
        if withdraw(i, v):
```

```
           deposit(j, v)
           unlock(i); unlock(j)
        else
           unlock(i); unlock(j)
           abort
```

General rule: Always acquire locks according to some consistent global ordering.

Why does this work? State of locks can be represented as directed graph. (the "Waits for" graph). Vertices represent transactions. Edge from vertex i to vertex j if transaction i is waiting for lock held by transaction j. Cycle in this graph indicates a deadlock.

Label the edge with its lock ID. For any cycle, there must be some pair of edges (i, j), (j, k) labeled with values m & n such that m > n. That implies that transaction j is holding lock m and it wants lock n, where m > n. That implies that j is not acquiring its lock in proper order.

This general scheme is known as two-phase locking.
More precisely, as strong strict two-phase locking.

General 2-phase locking
Phase 1. Acquire or escalate locks (e.g., read lock to write lock)
Phase 2. Release or deescalate locks

Strict 2-phase locking
During Phase 2. Release WRITE locks only at end of transactions

Strong strict 2-phase locking
During Phase 2. Release ALL locks only at end of transactions. This is the most common version. Required to provide ACID properties.

Other ways to handle deadlock

1. Have lock manager build waits-for graph. When it finds a cycle, chose an offending transaction and force it to abort.

2. Use timeout. Transactions should be short. If hit time limit, chose some transaction that is waiting for a lock and force it to abort.

Thinking about transactions.

For reliability, typically split transaction into phases:

1. Preparation. Figure out what to do and how it will change state,
   without altering state. Generate L: Set of locks, and U: List of updates
2. Commit or abort.
   a. If everything OK, then update global state.
   b. If transaction cannot be completed, leave global state
      unchanged.
   In either case, release all locks

Example:

```
xfer(i, j, v):
   L = {i, j}
   U = []  // List of required updates
   begin(L) // Begin transaction.  Acquire locks
   bi = Bal[i]
   bj = Bal[j]
   if bi >= v:
      Append(U, Bal[i] <- bi-v)
      Append(U, Bal[j] <- bj+v)
      commit(U, L)
   else
      abort(L)

commit(U, L):
        Perform all updates in U.
        Release all locks in L.

abort(L):
        Release all locks in L.
```

Part II Distributed Transactions

Same general idea, but state spread across multiple servers. Want to enable single transaction to read and modify global state and maintain ACID properties.

General idea:

1. Client initiates transaction. Makes use of "coordinator" (could be self).
2. All relevant servers operate as "participants".
4. The coordinator assigns a unique Transaction ID (TID) for the transaction.

Two phase commit:

Split each transaction into two phases:

1. Prepare & vote.
   Participants figure out all state changes
   Each determines if it will be able to complete transaction and communicates with coordinator

2. Commit.
   Coordinator broadcasts to participants whether to commit or abort
   If commit, then participants make their respective state changes

Implemented by set of messages between coordinator & participants:

1.A: Coordinator sends "CanCommit?" query to participants
  B: Participants respond with "VoteCommit" or "VoteAbort" to coordinator

2.A: If any participant votes for abort, the entire transaction must be aborted.
     Send "DoAbort" messages to participants. They release locks.
  B: Else, send "DoCommit" messages to participants. They complete transaction

Example. Suppose bank account i managed by server A, and account j by server B:

Then Server A would implement transaction:

```
L = {i}
begin(L)  // Acquire lock
U = []
b = Bal[i]
if b >= v:
   Append(U, Bal[i] <- b-v)
   vote commit
else vote abort
```

Server B would implement transaction

```
L = {j}
begin(L)  // Acquire lock
U = []
b = Bal[j]
Append(U, Bal[j] <- b+v)
vote commit
```

Server B can assume that there will be a big enough balance in i’s account. Entire transaction will abort otherwise.

What about locking?
Locks held by individual participants
* Acquired at start of preparation process
* Released as part of commit or abort.

Distributed deadlock:
* Possible to get cyclic dependency of locks by transactions across multiple servers
* Manifested in 2PC by having one of the participants unable to respond to a voting request (because it is still waiting to lock its local resources).
* Most often handle with timeout. Participant times out and then votes to abort. The transaction must then be retried.
  - Eliminates risk deadlock
  - Introduces danger of LIVELOCK: Keep retrying transaction but never succeed