Transactions without *Transactions*

Richard Kreuter and Kyle Banker
10gen Inc.

I. Preamble

II. Lit. Review

III. Requirements and Techniques

IV. Case Study

V. Tradeoffs, Future Work, Conclusion

I. Preamble

The *Hello World* of transactions
and why it's incomplete.

Transferring bank funds

- The classic textbook example that motivates ACID transactions is moving funds from one bank account to another.
  - In the thought experiment, both the debited fund and the credited fund exist inside the same DBMS.
  - By performing the debit and the credit within one hypothetical isolated transaction, the appearance of "conservation of money" is presented to clients.

But real transfers don't, and can't, work this way

- Real bank transfers aren't generally atomic with respect to concurrent account accesses (though they do preserve conservation of money).

- More generally, consider transfers between accounts that live in different banks. In these cases, there is no single transaction manager to isolate the transfer.

Terminology

- A ***database transaction*** is an atomic, durable, isolated set of state transformations in some kind of data store. (You all know what this means.)

- A ***transactional procedure*** is a process (e.g., a program or series of programs) that performs a set of state transformations in a manner that preserves invariants at the beginning and end of (but not necessarily during) its operations.

Database transactions have tradeoffs

- *Database transactions* are handy. (You know what we mean.)

- Too much reliance on *database transactions* encourages tight coupling between program and data stores, discourages modular program structure, and tends to impede scalability.

Transactional procedures have tradeoffs

- A *transactional procedure* decomposes problems into a set of atomic work units. This way, it models its problem after real-world processes and recovers gracefully from failures along the way. In addition, it permits the distribution of work across time and de-coupled systems.

- On the other hand, *transactional procedures* require careful coding and testing.

II. Lit. Review

Ideas for implementing transactions in a distributed and/or sloppy world.

Some things you should read

- Gregor Hohpe. ***Your Coffee Shop* Doesn't Use Two-Phase *Commit***. *IEEE Software*, pages 64–66,2005.
  - An analysis of the exchange between coffee drinkers, cashiers, and baristas, which turns out to be an asynchronous message queue with callbacks and compensatory transactions in exceptional cases.

- Hector Garcia-Molina, Kenneth Salem. ***Sagas***. SIGMOD Conference 1987. pp.249-259.
  - Introduces a model for long-lived transactions, possibly among multi-database systems, with slightly weakened ACID features but good scaling properties.

(And one more you might read)

- Jim Gray. ***The Transaction Concept: Virtues and Limitations*** Tandem Technical Report 81.3. June 1981
  - A high-level analysis of transactions, introduces the idea of the *compensatory transaction* as an application-level reaction to a failed (application-level) invariant.
  - For example, consider booking a multi-leg flight to be operated by different air carriers. From the application's perspective, the entire trip can only be booked if each leg can be booked. If, after booking some legs of the trip, the next leg fails, the application has two possible compensatory actions: cancel the reservations (and so fail the booking as a whole; a rollback), or retry the failed leg (possibly repeatedly, or with some other strategy; a roll forward).

III. Requirements and Techniques for Implementing Transactional Procedures

I. Requirements

A transactional procedure can be implemented with any data store having the following properties:

- Atomic **read-modify-write** semantics across a single unit of data.
- **Durable** writes.
- **Consistent** reads.
- Note: the application provides **soft isolation**.

II. Techniques

- Think of data as a **series of states**. Transition state forward, and build **compensation mechanisms** to cope with intermediate state change failures.
- Use a **transactional messaging** to communicate between systems (and elements of a saga).
- Think in terms of **transactional procedures** instead of single database transactions.

IV. A Case Study

Selling concert tickets
(i.e., managing limited inventory).

Selling Tickets: Problems

- No two seats are the same.
- Once we've guaranteed a particular seat, we must allow the user time to make the purchase.
- The user might not complete the purchase.
- The ticket must still be available at checkout.

Selling Tickets: Techniques

- Model the data after the real world: a physical record for every ticket.
- Implement a lease: once the user has chosen a seat, she has five minutes to purchase it.
- Use states and compensation mechanisms: the seat can be made available again if necessary.

Selling Tickets: Implementation

```
{ _id: ObjectId("4e76483349da435b640f50cf"),
  seat: 111,
  state: 'AVAILABLE'
}
```

Selling Tickets: Implementation

For each requested ticket:

1. Atomically fetch each ticket, testing for availability.

2. If available
    a. Set state to ‘IN-CART’
    b. Set order ID
    c. Set lease five minutes into the future.

Selling Tickets: Implementation

```
{ _id: ObjectId("4e76483349da435b640f50cf"),
  order_id: ObjectId("4e76495349da435b640f50d0"),
  seat: 111,
  state: 'IN-CART',
  expiration: Timestamp(1316374362)
}
```

Selling Tickets: Implementation

3. If any desired seat is unavailable, **compensate** by resetting all already-requested tickets to *AVAILABLE* state.

Selling Tickets: Implementation

User now has five minutes to pay. When the user initiates payment (clicks *PAY NOW*):

1. Atomically fetch each ticket, and test that the ticket is still *IN-CART* and not expired.

2. If so:
    a. Set state to ‘PRE-AUTHORIZE’
    b. Renew lease for five minutes into the future.
    c. Attempt to authorize credit card (external API)

Selling Tickets: Implementation

```
{ _id: ObjectId("4e76483349da435b640f50cf"),
  order_id: ObjectId("4e76495349da435b640f50d0"),
  seat: 111,
  state: 'PRE-AUTHORIZE',
  expiration: Timestamp(1316374662)
}
```

Selling Tickets: Implementation

4. a. If authorization fails, **compensate** by resetting tickets to *AVAILABLE*.

   b. If authorization succeeds, set tickets' state to *SOLD*.

Selling Tickets: Implementation

```
{ _id: ObjectId("4e76483349da435b640f50cf"),
  order_id: ObjectId("4e76495349da435b640f50d0"),
  seat: 111,
  state: 'SOLD',
  expiration: null
}
```

Selling Tickets: Notes

- We use a **procedural transaction** in lieu of a long-running transaction in one database.

- Therefore, we can run this in:
  - A distributed database or system.
  - A traditional database (without transactional overhead).

Selling Tickets: Notes

- We implement using a **lease** mechanism and a **series of states**.
- We store enough state to allow a **reaper process** to transition the database to an acceptable state in the event of a failure.
- A variation: selling a fixed number of general admission tickets would allow us to use **lazy expiration**.

V. Transactions without Transactions?

Advantages, disadvantages, and suggestions for further work.