# Distributed Transactions for Google App Engine

## Preliminary Report


- Daniel Shawcross Wilkerson - unemployed
- Simon Fredrick Vicente Goldsmith - Coverity
- Robert Johnson - SUNY Stony Brook
- Erick Armbrust - Google
- Ryan Barrett - Google

# Correctness & Performance

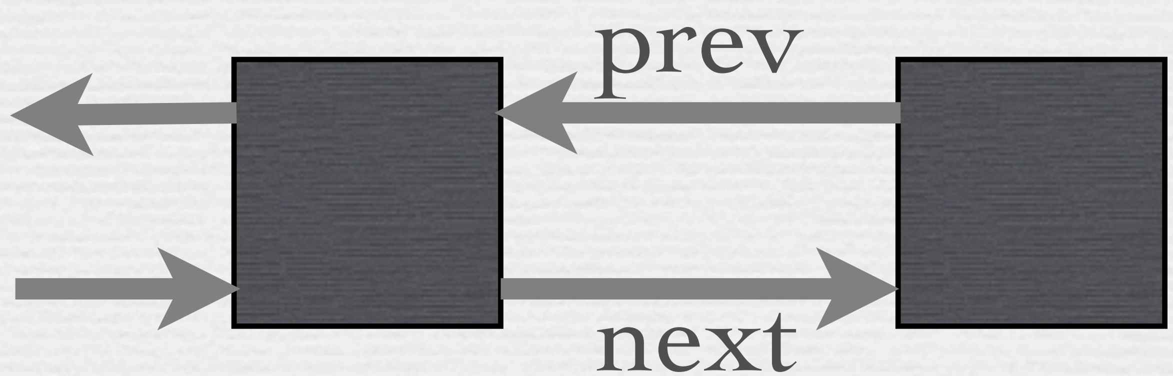
- **Correctness** and **performance** are the heart of engineering.
- **Correctness**: the output is what you want.
- **Performance**: the output doesn't cost too much.
    - Where cost is any resource: time, space, energy, money, people, machines.

# Correctness Requires *Invariants*

- Reasoning about program **correctness** requires **invariants.**
  - **Invariant**: a sentence that is always true; that which does not change when all else is changing.
- Use invariants from which you can ensure correctness.
  - **Initialize** invariants during construction;
  - **Maintain** them during operation.
- If you aren't thinking in terms of invariants, start now.
  - If you get nothing else out of my talk, remember this.

# Example Invariant: Data-structures

- A doubly-linked list module/class maintains the invariant that (credit: Scott McPeak):
  - **x->next == null** OR
  - **x->next->prev == x**




- Many other data-structures are similar.

# Example Invariant: Conservation of Money

- You are implementing a bank.
  - Alice transfers $20 to Bob.
- But no “money” is actually “transferred”:
  - numbers simply change within a machine.
- The **illusion** of **transferring** an **object** is maintained by several invariants, one being that
  - the **sum of all of the money does not change**.

# Scalability Requires *Distributed* Computing

- Unbounded **performance scalability** requires a large and therefore **distributed** computing machine.
  - "Small" machines give us the illusion of a single-point abstraction; this makes us lazy programmers.
- However, large/distributed machines are:
  - **non-reliable:** ongoing **random** local **failures**,
  - **non-serial: operate in** massive **parallel**, and
  - **non-synchronized: lack coordination** of behavior.
- **This is the future.**

# Distributed Computing makes Maintaining Invariants Hard

- Alice sends $20 to Bob.
    - Step 1: $20 added to Bob's account.
    - Process times out / **machine fails**....
    - The $20 was never subtracted from Alice's account.
- **Money has been created**
    - ... only the Federal Government can do that.

# *Transactions* Maintain Invariants

- (Correctness) Let a "**good**" state be one where all **invariants are satisfied**.
- (Performance) To make something happen, **invariants** often must be **temporarily violated**.
- Call a **set of operations** that take us **from one good state to another** a "**transaction**".

# ACID: The Correctness Perspective

- **Correctness:** program state stays within the subset of **good machine states.**
  - **Performance:** something has to happen.
- **Transactions jump** the machine **from state to state**:
  - **Durable**: **states persist**.
  - **Atomic** and **Isolated**: there are **no in-between states** for yourself or others.
  - **Consistent**: jump only from **good state to good state**.

# ACID in Detail

- **Durable**: once a transaction is done, need that the **changes persist**.
- **Atomic**: machines fail & processes time out, need that the **set of operations is all or nothing**.
- **Isolated**: need that **others see only states before or after** the transaction, not in the middle.
  - Isolated rather implies Atomic.
- **Consistent**: need that transactions only **go from one good state to another good state**.

# *Local* Transactions

- **If** transaction **data is localized**: gathered onto one machine, one locality within the distributed system,
  - **then** the **process is easier to control** and one may more easily implement the ACID properties.
- **Google App Engine** provides local txns:
  - at obj. construction time **objects may be grouped**;
  - a **txn may only operate on** the data of **one** **group**.
- **But then only *local* invariants can be maintained !**
- (Note that GAE is also strongly consistent.)

# Google App Engine

- **Google App Engine** allows people to build applications that scale arbitrarily.
- However applications have been able to ensure **local invariants**,
    - but **not global** ones...
- ***Until now.***

# Algorithm Overview

- **Run client**:
  - serve reads and record their version numbers;
  - buffer writes in shadow objects.
- **Get write locks** on written objects in key-order.
- **Check version** numbers of read objects;
  - also check they are not write locked.
- **Copy shadows** to their user objects in a local txn;
  - also update object version numbers and
  - delete write locks and shadows.

# Optional Cooperation

- These actions not needed for correctness, but should help reduce aborts due to contention.
- **Readers wait**: when read an object having a write lock, wait until the write lock is gone; otherwise we will abort later.
  - Simple and likely effective.
- **Writers wait**: before getting write locks, query for DTs reading it and wait if is unusually large; otherwise we abort them.
  - Needs calibration and may not be effective.

# This is Not as Easy As It Looks

- **Deadlock prevention**: holding locks creates a **waits-for graph**; a **cycle means no progress** will be made.
- **Ongoing progress**: a DT must **not languish** for lack of attention if it's **thread times-out**.
- **Concurrent roll-forward**: once past the client stage, other threads may have to **roll-forward a DT in parallel**; doing so must maintain correct operation.
- **Proof of Isolation**: guarantee that some serialization of the transactions is possible.

# Deadlock Prevention

- We use the traditional method of preventing deadlock: **lock the objects in** an order consistent with **a** shared **total order**.
  - We use the string ordering on the object keys.
- A DT can **only wait for locks later in the order** than the ones it already holds.
  - **Therefore** there can be **no cycle** in the wait-for graph.

# Ongoing Progress

- ... requires many subtleties to be handled correctly:
- **A read-storm (mostly) cannot keep out a write**: write lock taken but write not operated until reads complete. (A "read hurricane" will defeat it though.)
- **When blocked on another DT**, pause and **roll it forward**,
  - optimization: but only after it is more than about 10 seconds old (clocks are only approx. synch'ed).
- **The client should roll forward** DTs for **same user** before creating more DTs; we provide a way for that.
- We take care to **not create garbage**.

# Concurrent Roll-Forward

- In the **Client Operation** stage, the transaction is single-threaded and a timeout **aborts** it;
  - However **all later stages may be concurrently rolled-forward**.
- Therefore all stages of **getting locks, copying shadows, and releasing locks must be**
  - *Monotonic*: states progress in a sequence that can never go back.

# Monotonic Locking

- How to **get and release locks in monotonic way**?
  - **Getting** goes in one direction and **releasing** goes in another. <u>Different threads could fight over the lock</u>.
- **Example: three monotonic stages of writing**
  - **Create a shadow object** in the same entity group as the user object when the client writes.
  - **Get a write lock by writing the key** of the DT into the object to be locked.
  - **Copy and delete the shadow and delete the write lock** in one local transaction on the user object.

# Need Strong Consistency

- Vogels: "**Eventual consistency**: the storage system guarantees that if no new updates are made to the object, **eventually all accesses will return the last updated value**".
- Vogels: "**Strong consistency**: After the update completes, **any subsequent access** [by any process in the system] **will return the updated value**." We need that.

# Saving Users from Themselves

- **Distributed and Local txns don't mix**. Erick and Ryan require an object to be DT or LT flavored.
  - Local txns don't honor the Distributed txn locks.
- **Read then Write then Stop**: client reads x=1, writes x=2, then reads x again; writes are buffered, so the client will read x=1 !
  - Thus, allow neither reads nor writes after a write.
- **Did my DT finish?**: the client must query for any timed-out DTs and first roll them forward.
  - User can query for async complete/abort DTs.

# Queries Are Not Handled

- We provide **no transactional semantics for queries,**
  - **only** for **reads and writes** on sets of objects **specified directly by the object key**.
- In Google App Engine, **queries can return objects that do not satisfy the query** and can fail to return objects that do satisfy the query.
  - Modulo that, you are free to make a query and then mark as read all object returned from a query.
- Further work on queries is needed.  Providing transactional semantics is very predicate-dependent.

# ACID Correctness Example, Isolation

- The hardest part of proving that we provide the ACID properties is showing Isolation.
- Example:
  - **DT1** has write locks on objects *A* and *B* and **has written *A* but not yet *B*.**
  - **DT2 reads both** *A* and *B* -- **an inconsistent read!**
- As Simon says, "**we want to make sure that DT2 is doomed**".

# An Inconsistent Read is Doomed (cont.)

- DT1 has written A but not B; DT2 has read A and B.
  - DT2 has an inconsistent read and we want it to fail.
- **If DT1 completes first**
  - when **DT2** tries to complete, it **will fail** when doing the **version number check** on B.
- **If DT2 tries to complete first**
  - it cannot because **DT1 still has the write locks** on A and B and will fail the write lock check.

# Detailed Example Transcript

Continuing the previous example, here is a possible transcript.

Initial: Alice, balance: $200; Bob, balance: $100.

DT1: Client Operation start: transfer $20 from Alice to Bob.
DT1: Client Operation done.
DT1: Get Locks.
DT1: Read Check (no changes written yet...).

DT2: Client Operation start: transfer $190 from Alice to Bob.
DT2: Alice, balance: $200; Bob, balance: $100.
DT2: Client Operation done.

DT1: Copy Shadows (incl. updating version nums) and DONE.

DT2: Get Locks.
DT2: Read Check version number check fails; ABORT!

Final: Alice, balance: $180; Bob, balance: $120.

# Future Work

- Queries within transactions?
    - Locking is very dependent upon the predicate.
- Do we need the underlying layer to be strongly consistent?
- Performance?
    - Deep integration with GAE/Big Table infrastructure?

# Conclusion

- Distributed Transactions on Google App Engine exist.
- Let us know if they help you.