Putting Consistency Back into Eventual Consistency

Presented by David Lang

Overview

- Terminology and definitions
- Explicit Consistency
- Invariants
- Restricting concurrency using Reservations
- Implementation of Indigo
- Evaluations

Distributed System Review

- Maintaining copies of application data across multiple physical locations and machines
- Avoid overhead and latency issues through geo-replication
- Systems rely on consistency definitions depending on how strict certain constraints need to be
  - Weaker consistency can lead to non-intuitive semantics
  - Stronger consistency can be tricky to maintain and less fault tolerant
- Indigo enforces Explicit Consistency between geo-replicated servers

Definitions

- t(S) - state after the write operation of some commit transaction t to state S
- S_n - the n^{th} state of the database after transactions $t_1, \ldots, t_n$
- T(S) - set of transactions included in S
- $t_a \prec t_b$ - t_a executed before t_b
- $t_a \parallel t_b$ - t_a and t_b are concurrent
- $O = (T, \prec)$ - $\prec$ defines a partial order in T
  - $O' = (T, <)$ - O' is a valid serialization of $O = (T, \prec)$ if O' is a linear extension of O

Explicit Consistency

- Programmers explicitly define application-specific correctness rules
  - Expressed as **invariants** over the database state
  - Given that each machine locally tracks its own invariants, concurrent updates still may cause violations
- Invariant denoted as *I*
- S is an *I-valid state* if *I* holds in S, e.g. *I*(S) = true
- Given T and $\lessdot$, O_i = (T, <) is an *I*-valid serialization of O = (T, $\lessdot$) if O_i is a valid serialization of O, and *I* holds in every state that results from executing some prefix of O_i
- **Explicit Consistency**: All serializations of O = (T, $\lessdot$) are *I*-valid

Explicit Consistency Approach

1. Detect the set of operations that may lead to invariant violation when executed concurrently
    a. These operations form the *I-offender* set
    b. Programmers perform static analysis by comparing the effects of operations/transactions and existing invariants to find possible *I-offenders*
2. Find an efficient mechanism for handling these *I-offender* sets (violence avoidance)
    a. Restrict concurrency to avoid invariant violations
    b. Use either conflict-repair or conflict-avoidance measures
3. Form application code to use the selected mechanism
    a. Involves extending existing operations
    b. Call to the API functions on **Indigo**

Determining *I*-Offender Sets

- Defining invariants:
  - Application invariants can be represented by first-order logic
  - Can express a wide variety of consistency constraints, including set membership and numerical constraints - SMT solving!
  - ex. Player(P), Enrolled(P, T), nrPlayers (T) ≤ 5.
- Defining operational postconditions:
  - Two types of side-effect clauses for after an operation:
    - Predicate clause: describes the truth assignment for some predicate
      - removePlayer(P) has a postcondition that ¬Player(P)
    - Function clause: describes the relation between the initial and final function values
      - enrollTournament(P, T) has a postcondition where enrolled(P, T), nrPlayers(T) = nrPlayers (T) + 1

Determining *I*-Offender Sets

- Restricting states:
  - Numeric constraints: Setting upper/lower bounds on data values
    - ex. $\forall P, player(P) \Rightarrow budget(P) \geq 0$
  - Integrity constraints: Specifies relations between different objects
    - ex. $\forall P, hasScore(P) \Rightarrow player(P)$
  - General state constraints: Constraining certain aspects of the current state
- Restricting state transitions:
  - Implemented through setting an invariant over the state of the database
  - Uses a *history variable* that records its state at some point in time
  - ex. $\forall P, T, active(T) \wedge enrolled(P, T) \Rightarrow participant(P, T)$

Determining *I*-Offender Sets

- Existential quantifiers: defines “ground rules”
  - ex. $\forall$ T, tournament(T) $\Rightarrow$ $\exists$ P, enrolled(P, T)
- Uninterpreted predicates and functions
  - Imposes limitations to the invariants that can be expressed
  - Programmers must formulate broader statements over the database

Algorithm 1 Algorithm for detecting unsafe operations.

Require: I : invariant; O : operations.

1: $C \leftarrow \emptyset$ {subsets of unsafe operations}
2: **for** $op \in O$ **do**
3: **if** *self-conflicting*$(I, \{op\})$ **then**
4: $C \leftarrow C \cup \{\{op\}\}$
5: **for** $op, op' \in O$ **do**
6: **if** *opposing*$(I, \{op, op'\})$ **then**
7: $C \leftarrow C \cup \{\{op, op'\}\}$
8: **for** $op, op' \in O : \{op, op'\} \notin C$ **do**
9: **if** *conflict*$(I, \{op, op'\})$ **then**
10: $C \leftarrow C \cup \{op, op'\}\}$
11: **return** C

addPlayer(P) → Postcondition: Player(P)

removePlayer(P) → Postcondition: ¬Player(P)

Handling *I*-Offender Sets

- Invariant repair
  - Allow conflicting operations to execute concurrently, then fix the violation after the fact
  - **Indigo** only has limited support for this method since it can only address invariants on single database objects
- Invariant-violation avoidance
  - Avoid violations by combining the effects on invariants

Reservations

- *shared forbid*, giving the shared right to forbid some action to occur
- *shared allow*, giving the shared right to allow some action to occur
- *exclusive allow*, giving the exclusive right to execute some action

Restricting Concurrency by Reservations

- UID Generator
  - Gives uniqueness to identifiers by splitting the space of identifiers between all replicas
  - Prevents overlap of IDs
- Multi-level lock reservation
  - Multiple levels of restrictions: shared forbid, shared allow, exclusive allow
- Multi-level mask reservation
  - Effectively a vector of multi-level locks
  - For invariants $P_1 \vee P_2 \vee \ldots \vee P_n$, a mask of length n is created to assign locks to each invariant
- Escrow reservation
  - Checking a count of elements against some condition
- Partition lock reservation
  - A lock to reserve a part of a partitionable resource

Implementation of Indigo

- This paper's implementation of an API that enforces Explicit Consistency through reservations
  - Reservations are stored as objects
  - Each reservation object keeps track of rights assigned to each replica
- Middleware prototype implemented on top of geo-replicated datastores
  - Supports causal consistency, transactions that merge updates from a snapshot database, and linearizable execution of operations
  - Reservation manager is implemented in each replica to exchange and receive reservations
  - Supported by Indigo API calls
- Fault tolerance maintained by underlying storage system
- Z3 used as the SMT solver for constraining reservations and finding *I*-offender sets

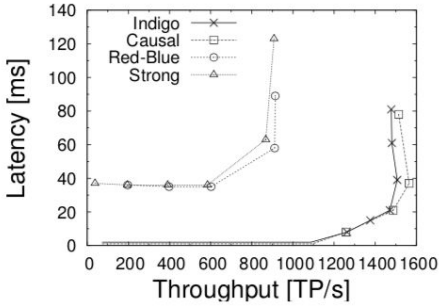


Figure 2. Peak throughput (ad counter application).

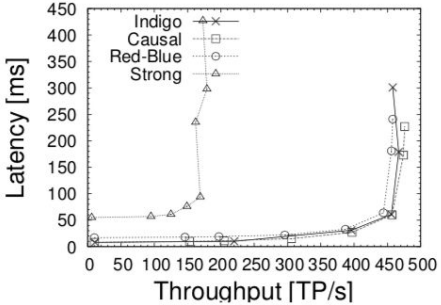


Figure 3. Peak throughput (tournament application).

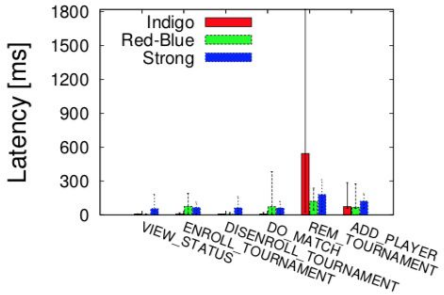
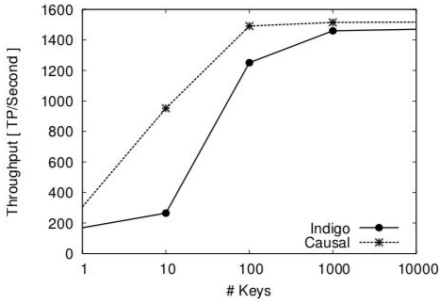
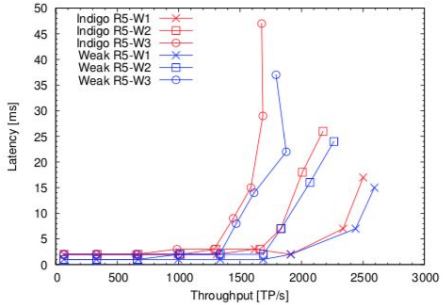


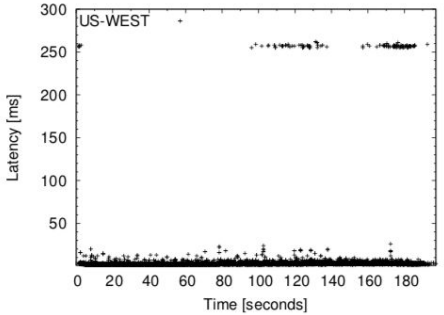
Figure 4. Average latency per op. type - Indigo (tournament app.).



(a) Peak throughput with increasing contention (ad counter application).



(b) Peak throughput with an increasing number of invariants (ad counter application).



(c) Latency of individual operations of US-W datacenter (ad counter application).

Figure 5. Micro-benchmarks.

Discussion Questions

- While Indigo provides an effective way to enforce Explicit Consistency, communicating reservations between distributed servers still relies on availability between the replicas. What are some of the bottlenecks when reservations fail to be consistently retrieved?

- The prototype for Indigo, at its base, was built with only causal consistency guaranteed. Can we guarantee stronger consistency models and have the implementation be valid?