Granola: Low-Overhead Distributed Transaction Coordination

James Cowling and Barbara Liskov
MIT CSAIL
Granola

An infrastructure for building distributed storage applications.

A distributed transaction coordination protocol, that provides strong consistency without locking.
Distributed Storage

Repository 1

Repository 2

Repository 3

Client 1

Client 2

...
Why Transactions?

**Atomic operations** allow users and developers to ignore concurrency.

Distributed atomic operations allow data to **span multiple repositories** and avoid **inconsistency** between repositories.
Distributed transactions are hard

Tension between consistency and performance
Opting for Consistency

Two-phase commit with strict two-phase locking

Distributed databases, Sinfonia, etc.
Opting for Consistency

Two-phase commit with strict two-phase locking.

Distributed databases, Sinfonia, etc.

High transaction cost: multiple message delays, forced log writes, locking/logging overhead (≈30-40%*)

* OLTP through the looking glass, and what we found there
Harizopoulos et al, SIGMOD ‘08
Opting for Performance

No distributed transactions
SimpleDB, Bigtable, CRAQ, etc.

Weak consistency models
Dynamo, Cassandra, MongoDB, Hbase, PNUTS, etc.
Opting for Performance

No distributed transactions

SimpleDB, Bigtable, CRAQ, etc.

Weak consistency models

Dynamo, Cassandra, MongoDB, Hbase, PNUTS, etc.

Place the burden of consistency on the application developer
Where we come in...

Strong Consistency

and

High Performance

(for a large class of transactions)
Introduce a new transaction class which lets us provide consistency without locking
MOTIVATION

TRANSACTION MODEL

PROTOCOL

EVALUATION
One-Round Transactions

Expressed in one round of communication between clients and repositories

Execute to completion at each participant
General Operations

Transactions are uninterpreted by Granola, and can execute arbitrary operations.
Transaction Classes

Single-Repository

execute entirely on one repository

Distributed Transactions

• Coordinated

• Independent
Coordinated Transactions

Commit only if all participants vote to commit

Example:

• Transfer $50 between accounts
Independent Transactions

Transactions where all participants will make the same commit/abort decision

Examples:

• Add 1% interest to each bank balance.
• Compute total amount of money in the bank.
Independent Transactions

Evidence these are common in OLTP workloads

- Any read-only distributed transaction
- Transactions that always commit
- Atomically update replicated data
- Where commit decision is deterministic function of shared state
Example: TPC-C

TPC-C benchmark can be expressed entirely using single repository and independent transactions

e.g., new_order transaction only aborts if invalid item number can be computed locally if we replicate Item table
Motivation
Transaction Model
Protocol
Evaluation
Replication

Repository

Implemented as

Primary

Backup

Backup
Repository Modes

Primarily in **Timestamp Mode**

- Single-repository, Independent

Occasionally in **Locking Mode**

- When coordinated transactions are required
Timestamps

Each transaction is assigned a timestamp

Each repository executes transactions in timestamp order

Timestamps define global serial order
Key Questions

How do we assign timestamps in a scalable, fault-tolerant way?

How do we make sure we always execute in timestamp order?
Single-Repository Protocol

Clients present the highest timestamp they have observed

Repository chooses timestamp higher than the client timestamp, any previous transaction, and its clock value

Repository executes in timestamp order, sends response and timestamp to client
Assign timestamp

Log Transaction

Run

Choose timestamp higher than previous transactions
Assign timestamp

Log Transaction

Run

Choose timestamp higher than previous transactions

Run replication protocol to record timestamp
Assign timestamp

Choose timestamp higher than previous transactions

Log Transaction

Run replication protocol to record timestamp

Run

Execute in timestamp order
Send result and timestamp to the client
Independent Protocol

Clients present highest timestamp they observed

Repository chooses proposed timestamp higher than clock value and previous transactions

Repositories vote to determine highest timestamp

Repository executes in timestamp order, sends timestamp to client
Propose timestamp

Log Transaction

Vote

Pick final timestamp

Run

Choose **proposed timestamp** higher than previous transactions

Run replication protocol to record proposed timestamp

Send proposed timestamp to the other participants

**Highest timestamp** from among votes

Execute in timestamp order
Send result and final timestamp to client
Timestamp Constraint

Won’t execute transaction until it has the lowest timestamp of all concurrent transactions

Guarantees a global serial execution order
Example 1:

Queue: 
History: 

Repository 1

Queue: 
History: 

Repository 2

Alice

Bob
Queue:  T1 [prop. ts: 9]  Queue:  T1 [prop. ts: 3]
History:  History:  

Repository 1  

Vote T1 [9] ...  Vote T1 [3]

Repository 2  

Alice  

Bob
Queue: 

History: T1 [final ts: 9]

Queue: T1 [prop. ts: 3]

History: 

Repository 1

Vote T1 [9] ...

Repository 2

T2

Alice

Bob
Queue: T1 [prop. ts: 3], T2 [ts: 5]

History: T1 [final ts: 9]

Repository 1

Vote T1 [9] ...

Repository 2

T2

Alice

Bob
Queue: T1 [prop. ts: 3], T2 [ts: 5]

History: T1 [final ts: 9]

Queue: 

History: 

Vote T1 [9] ...

Repository 1 

Repository 2

Alice 

Bob
Queue: T1 [prop. ts: 3], T2 [ts: 5]
Queue: 
History: T1 [final ts: 9]
History: 

Vote T1 [9]

Repository 1

Repository 2

Alice

Bob
Queue: T2 [ts: 5], T1 [final ts: 9]

History: T1 [final ts: 9]

Vote T1 [9]

Repository 1

Repository 2

Alice

Bob
Queue:

History:
T1 [final ts: 9]

Repository 1

Queue:

History:
T2 [ts: 5], T1 [final ts: 9]

Repository 2

T1 [final ts: 9]

Alice

Bob
Queue: [ ]
History: T1 [final ts: 9]

Queue: [ ]
History: T2 [ts: 5], T1 [final ts: 9]

Global serial order: T2 -> T1

Repository 1

Repository 2

Alice

Bob
Choosing timestamps

Client-provided timestamp guarantees transaction will be serialized after any transaction it observed
Example 2:

Queue: T1 [final ts: 9]

History: T1 [final ts: 9]

Queue: T1 [prop. ts: 3]

History: 

Vote T1 [9] ...

Repository 1

Repository 2

Alice

Bob
Queue: 
History: T1 [final ts: 9]

Queue: T1 [prop. ts: 3]
History: 

Vote T1 [9] ...

Repository 1

T2

Repository 2

Alice

Bob
Queue:

History: T1 [final ts: 9], T2 [ts: 10]

Queue: T1 [prop. ts: 3]

History: 

T1 [ts: 9] ...

Vote T1 [9] ...

Repository 1

Alice

Bob

Repository 2

T2 [ts: 10]
Queue: T1 [prop. ts: 3], T3 [ts: 11]

History: T1 [final ts: 9], T2 [ts: 10]

Vote T1 [9] ...

Repository 1

Repository 2

T3 [latest ts: 10]

Alice

Bob
Queue: T1 [final. ts: 9], T3 [ts: 11]

History: T1 [final ts: 9], T2 [ts: 10]

Queue: [empty]

History: [empty]

Repository 1

Vote T1 [9]

Repository 2

Alice

Bob
Queue: 

History: T1 [final ts: 9], T2 [ts: 10]

Queue: 

History: T1 [final ts: 9], T3 [ts: 11]

Global serial order: T1 -> T2 -> T3
Where are we now?

Timestamp-based transaction coordination

- Interoperability with coordinated transactions
- Recovery from failures
Coordinated Transactions

Application determines commit/abort vote

Requires locking to ensure vote isn’t invalidated
Protocol changes

Prepare phase
application acquires locks and determines vote

Timestamp voting

Repository can commit transactions out of timestamp order
Protocol changes

Prepare phase where application acquires locks and determines vote

Repository can commit transactions out of timestamp order

**Timestamp Constraint:** Timestamps *still* match the serial order, even if execution happens out of timestamp order
Locking Mode

Repository 1

Client 1

Repository 2

Client 2

Repository 3

Timestamp Mode

coordinated transaction

independent transaction
Effect on other Transactions

Independent transactions processed using locking-mode protocol

**No locks held** for single-repository transactions
MOTIVATION

TRANSACTION MODEL

PROTOCOL

EVALUATION
Experimental Setup

Implemented as a Java library

Deployed on 20 servers
- 2005-vintage Xeon machines
- Colocated on same LAN
Experimental Setup

Throughput was CPU bound

> 65,000 tps on microbenchmarks
latency kept around 1-2 ms

We compare against extended version of Sinfonia
TPC-C

Implemented distributed TPC-C benchmark using a non-distributed codebase

Partitioned such that all transactions single-repository or independent
Scalability

Total Throughput (tps)

Granola
Granola (locking)
Sinfonia

Repositories

1 2 3 4 5 6 7 8 9 10
Distributed transactions

Graph showing throughput (tps) vs. distributed transactions. The graph compares Granola, Granola (locking), and Sinfonia.
Performance Gains

No locking or undo logging
40% lower CPU overhead
no aborts or retries

3 one-way message delays (vs 4)

1 forced log write
Conclusion

Granola provides strong consistency and high performance for distributed transactions.

Exploit independent transactions to provide coordination without locking.