Transactions in RAMCloud

Diego Ongaro

Stanford University

RAMCloud Design Review
April 1, 2010

Where's my CMPXCHG?

We can't even increment a value safely with our poor API.

Example: Unsafe Increment

```
data = ramcloud.read(table, 10)
ramcloud.write(table, 10, data + 1)
```

Problem: Race condition

- If you're surprised, Professor Ousterhout is teaching the introductory operating systems course this quarter.

What primitives do apps need for concurrency?

Hypothesis:
If latency is sufficiently low, can provide a high level of consistency

- Can push complex object operations to apps
- A lot of other NoSQL systems don't do this

Outline

1. Conditional operations for single objects
2. Transactions for multiple objects

RAMCloud's Conditional API

- Each object has a monotonically increasing version number
- Predicates specify whether an object must exist
  and whether it must have a given version number

```
read(table ID, object ID, predicates) → data, version
write(table ID, object ID, predicates, data) → version
delete(table ID, object ID, predicates)
```

Example: Atomic Increment

```
label .again:
  data, v1 = ramcloud.read(table, 10, None)
  try:
      v2 = ramcloud.write(table, 10, Pred(version=v1), data + 1)
  except: # Someone else changed the object first!
      goto .again
```

Transactions Are a Useful Building Block

- Apps may need to simultaneously update multiple objects
  - Transferring money across users
  - Updating a shared data structure

- Transactions make this easy (well, relatively)
  - Apps can maintain database invariants

- Alternatives are too difficult
  - Locking isn't an option – we can't detect when apps crash
  - Expired leases (locks with timeouts) are difficult to clean
  - Lockless data structures are tricky

Optimistic Concurrency Control

- Transactions proceed without locking
- During commit, make sure the objects read have not changed

We expect few conflicts:

- Writes are rare
- Transactions are rarer
  - Some apps won't need them
  - Most writes can use conditional API
- Conflicts are rarer yet
- High speed $\Rightarrow$ fewer conflicts

Optimistic Transactions API

Minitransaction – packaged set of conditional operations to execute atomically

- Modeled after Sinfonia

Optimistic Transactions API

Minitransaction – packaged set of conditional operations to execute atomically

- Modeled after Sinfonia

Example: Move $20 from account 1 to account 2

```
label .again:
  tx = ramcloud.Transaction()
```

Minitransaction

Object	Pred	Operation

Optimistic Transactions API

Minitransaction – packaged set of conditional operations to execute atomically

- Modeled after Sinfonia

Example: Move $20 from account 1 to account 2

```
label .again:
  tx = ramcloud.Transaction()
  d1, v1 = tx.read(ACC, 1)
```

Minitransaction

Object	Pred	Operation
ACC: 1	v_1	

Optimistic Transactions API

Minitransaction – packaged set of conditional operations to execute atomically

- Modeled after Sinfonia

Example: Move $20 from account 1 to account 2

```
label .again:
  tx = ramcloud.Transaction()
  d1, v1 = tx.read(ACC, 1)
  tx.write(ACC, 1, d1 - 20)
```

Minitransaction

Object	Pred	Operation
ACC: 1	v_1	write($d_1 - 20$)

Optimistic Transactions API

Minitransaction – packaged set of conditional operations to execute atomically

- Modeled after Sinfonia

Example: Move $20 from account 1 to account 2

```
label .again:
  tx = ramcloud.Transaction()
  d1, v1 = tx.read(ACC, 1)
  tx.write(ACC, 1, d1 - 20)
  d2, v2 = tx.read(ACC, 2)
```

Minitransaction

Object	**Pred**	**Operation**
ACC: 1	v_1	write$(d_1 - 20)$
ACC: 2	v_2	

Optimistic Transactions API

Minitransaction – packaged set of conditional operations to execute atomically

- Modeled after Sinfonia

Example: Move $20 from account 1 to account 2

```
label .again:
  tx = ramcloud.Transaction()
  d1, v1 = tx.read(ACC, 1)
  tx.write(ACC, 1, d1 - 20)
  d2, v2 = tx.read(ACC, 2)
  tx.write(ACC, 2, d2 + 20)
```

Minitransaction

Object	**Pred**	**Operation**
ACC: 1	v_1	write($d_1 - 20$)
ACC: 2	v_2	write($d_2 + 20$)

Optimistic Transactions API

Minitransaction – packaged set of conditional operations to execute atomically

- Modeled after Sinfonia

Example: Move $20 from account 1 to account 2

```
label .again:
  tx = ramcloud.Transaction()
  d1, v1 = tx.read(ACC, 1)
  tx.write(ACC, 1, d1 - 20)
  d2, v2 = tx.read(ACC, 2)
  tx.write(ACC, 2, d2 + 20)
  try:
      tx.commit()
  except:
      goto .again
```

Minitransaction

Object	Pred	Operation
ACC: 1	v_1	write($d_1 - 20$)
ACC: 2	v_2	write($d_2 + 20$)

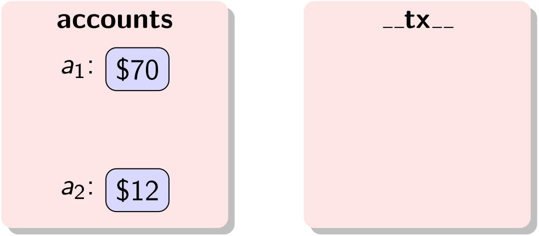
Approaches

1. Client-Side Transactions
   - No server modifications required – built on the conditional API
2. Two-Phase Commit (2PC)
   - Better performance

Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

Transferring \$20 from a_1 to a_2:

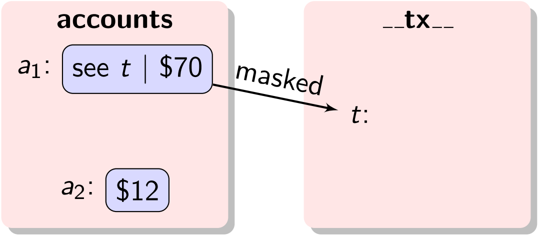


Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

1. Mask accounts to super-object t (in any order)
   - t, if it exists, in effect *contains* all of its masked objects

Transferring \$20 from a_1 to a_2:

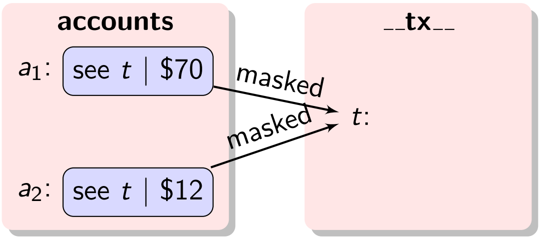


Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

1. Mask accounts to super-object t (in any order)
   - t, if it exists, in effect *contains* all of its masked objects

Transferring \$20 from a_1 to a_2:

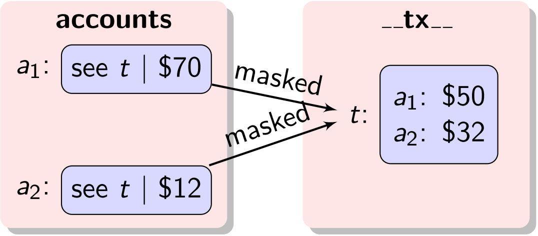


Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

1. Mask accounts to super-object t (in any order)
   - t, if it exists, in effect *contains* all of its masked objects
2. Fill in t – **this is the commit**

Transferring \$20 from a_1 to a_2:

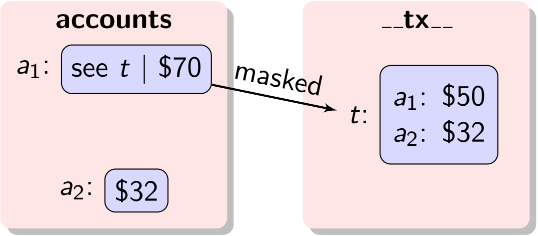


Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

1. Mask accounts to super-object t (in any order)
   - t, if it exists, in effect *contains* all of its masked objects
2. Fill in t – this is the commit
3. Write back values, unmask accounts (in any order)

Transferring $20 from a_1 to a_2:

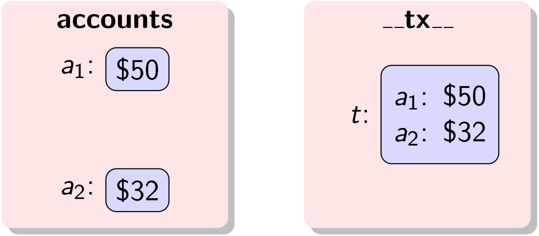


Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

1. Mask accounts to super-object t (in any order)
   - t, if it exists, in effect *contains* all of its masked objects
2. Fill in t – this is the commit
3. Write back values, unmask accounts (in any order)

Transferring $20 from a_1 to a_2:

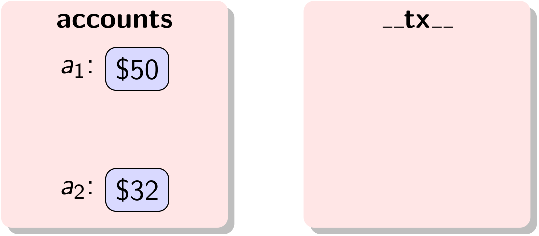


Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

1. Mask accounts to super-object t (in any order)
   - t, if it exists, in effect *contains* all of its masked objects
2. Fill in t – this is the commit
3. Write back values, unmask accounts (in any order)
4. Delete t

Transferring \$20 from a_1 to a_2:



Client-Side Transactions

Idea: Put the values together into a single object which we can update atomically.

1. Mask accounts to super-object t (in any order)
   - t, if it exists, in effect *contains* all of its masked objects
2. Fill in t – this is the commit
3. Write back values, unmask accounts (in any order)
4. Delete t

Optimization

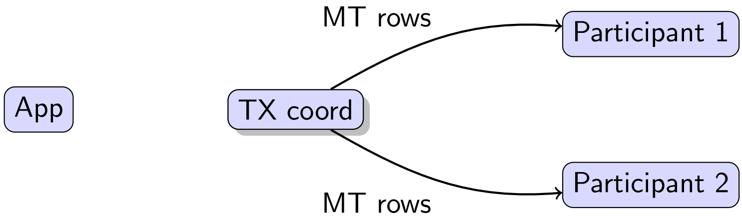
- Server-side changes for cheap masking
  - Saves rewriting the data to the log in step 1

Two Phase Commit

1. App sends MT to transaction coordinator (a participant)

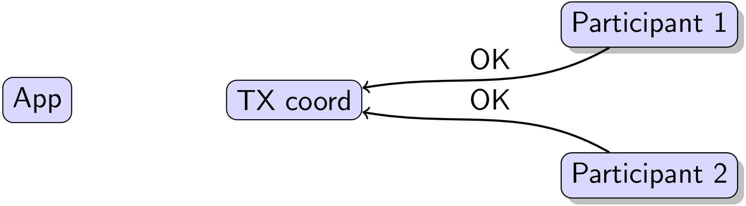
Two Phase Commit

1. App sends MT to transaction coordinator (a participant)
2. Coord logs participant list, sends MT rows to participants



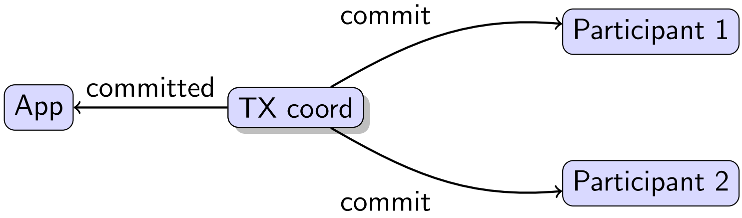
Two Phase Commit

1. App sends MT to transaction coordinator (a participant)
2. Coord logs participant list, sends MT rows to participants
3. Participants lock objects, log MT rows, send vote to coord



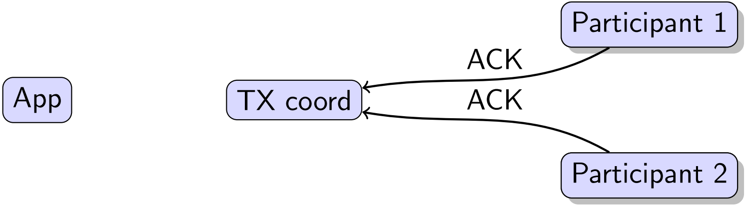
Two Phase Commit

1. App sends MT to transaction coordinator (a participant)
2. Coord logs participant list, sends MT rows to participants
3. Participants lock objects, log MT rows, send vote to coord
4. Coordinator logs decision, sends to participants and app



Two Phase Commit

1. App sends MT to transaction coordinator (a participant)
2. Coord logs participant list, sends MT rows to participants
3. Participants lock objects, log MT rows, send vote to coord
4. Coordinator logs decision, sends to participants and app
5. Participants commit MT rows, send ack to coordinator



Two Phase Commit

1. App sends MT to transaction coordinator (a participant)
2. Coord logs participant list, sends MT rows to participants
3. Participants lock objects, log MT rows, send vote to coord
4. Coordinator logs decision, sends to participants and app
5. Participants commit MT rows, send ack to coordinator
6. Coordinator cleans log entries

Two Phase Commit

1. App sends MT to transaction coordinator (a participant)
2. Coord logs participant list, sends MT rows to participants
3. Participants lock objects, log MT rows, send vote to coord
4. Coordinator logs decision, sends to participants and app
5. Participants commit MT rows, send ack to coordinator
6. Coordinator cleans log entries

Optimizations

- App acts as transaction coordinator
  - App/coordinator does not log
  - Participant list replicated on all participants instead

Client-Side vs 2PC Comparison

Bytes written to the log:

- Client-side: 3x (mask, super-object, write-back)
- Client-side with server mods: 2x (super-object, write-back)
- 2PC flavors: 1x

Serial RPCs for app to resume processing, including log appends:

- Client-side flavors: 4 (mask, super-object)
- 2PC: 5
- 2PC with app coordinating: 2

What's this hiding?

- Client-side flavors require weak access control
  - Apps write back values for others' crashed transactions
- Complexity

Conclusion

- Low latency affords us high consistency
- Conditional API provides atomic ops for a single object
- Optimistic transactions for multiple objects
  - Optimized client-side approach about twice as slow as 2PC
  - We haven't decided on an approach, may try both

Questions/Comments

Some for the audience:

- Do applications need transactions?
- Are conflicts as rare as we hope?
- Client-side or 2PC?