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

```
d1 = ramcloud.read(1, 10)
d2 = str(int(d1) + 1)
ramcloud.write(1, 10, d2)
```

Problem: Race condition

RAMCloud's Conditional API

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

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

Example: Atomic Increment

```
label .again:
    d1, v1 = ramcloud.read(1, 10, None)
    d2 = str(int(d1) + 1)
    try:
        v2 = ramcloud.write(1, 10, Predicates(version=v1), d2)
    except: # Someone else changed the object first!
        goto .again
```

Transactions Are a Useful Building Block

RAMCloud provides basic primitives: tables and objects

- ...and maybe some more complex functionality: indexes

Clients must build concurrent data structures out of these

- May need to simultaneously update multiple objects
  - Splitting nodes in a B^+-tree
  - Transferring assets across users
  - Friending someone on Facebook
- Transactions make this easy (well, relatively)
  - Apps can maintain database invariants
- Alternatives are too difficult
  - Locking isn't an option – apps might crash
  - Lockless data structures are tricky
  - Expired leases are difficult to clean

Optimistic Concurrency Control

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

Approaches

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

Optimistic Transactions API

Transferring \$20 from a_1 to a_2.

```
label .again:
  tx = ramcloud.Transaction()
  a1, v1 = tx.read(ACC, 1)
  a2, v2 = tx.read(ACC, 2)
  a1m = str(int(a1) - 20)
  a2m = str(int(a2) + 20)
  tx.queueWrite(1, 10, a1m)
  tx.queueWrite(1, 20, a2m)
  try:
      tx.commit()
  except:
      goto .again
```

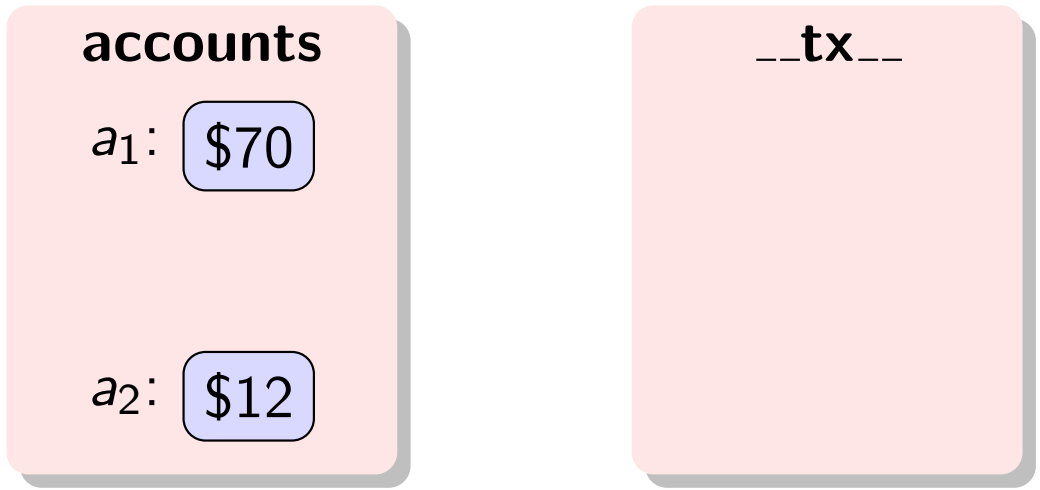
Minitransaction

Object	Pred	Op
ACC: 1	v1	write(a1m)
ACC: 2	v2	write(a2m)

TODO: build up code and minitransaction incrementally

Client-Side Transactions

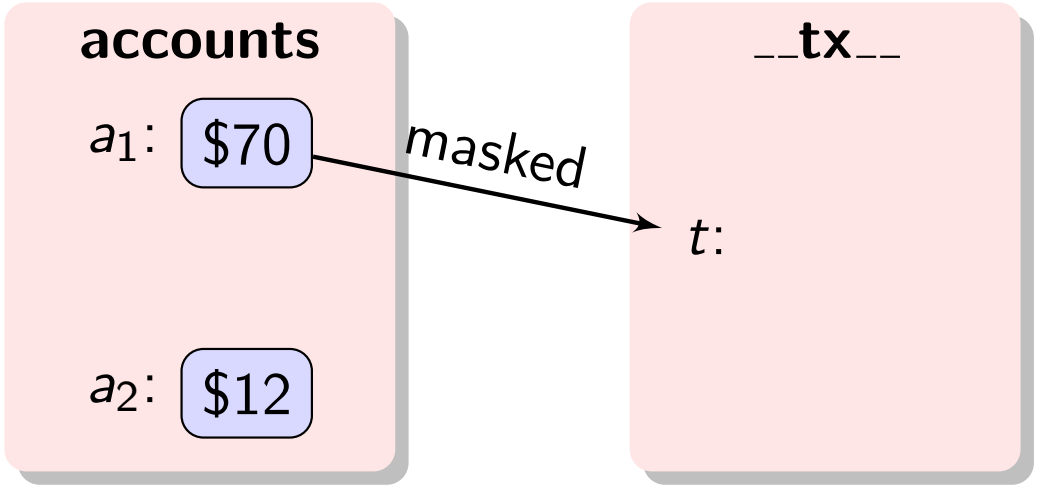
Transferring \$20 from a_1 to a_2:



Client-Side Transactions

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

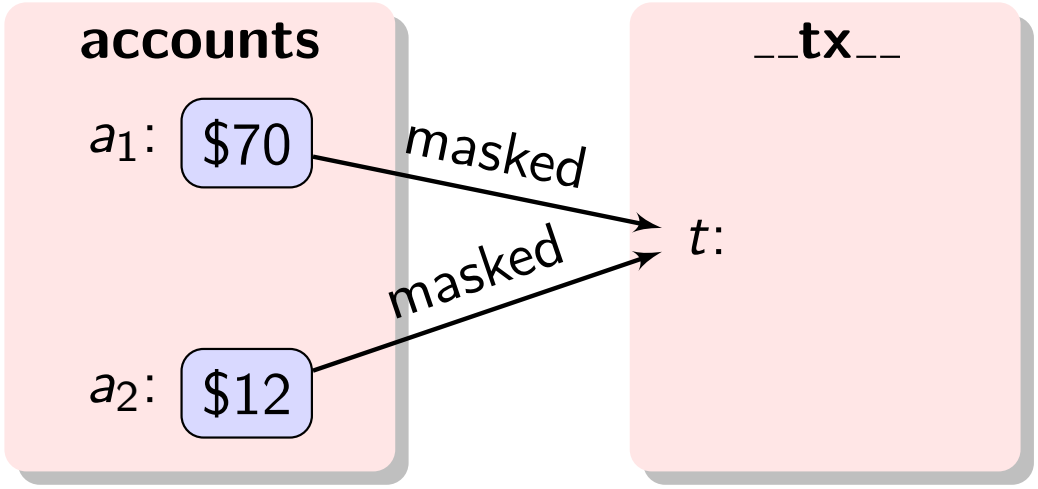
Transferring \$20 from a_1 to a_2:



Client-Side Transactions

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

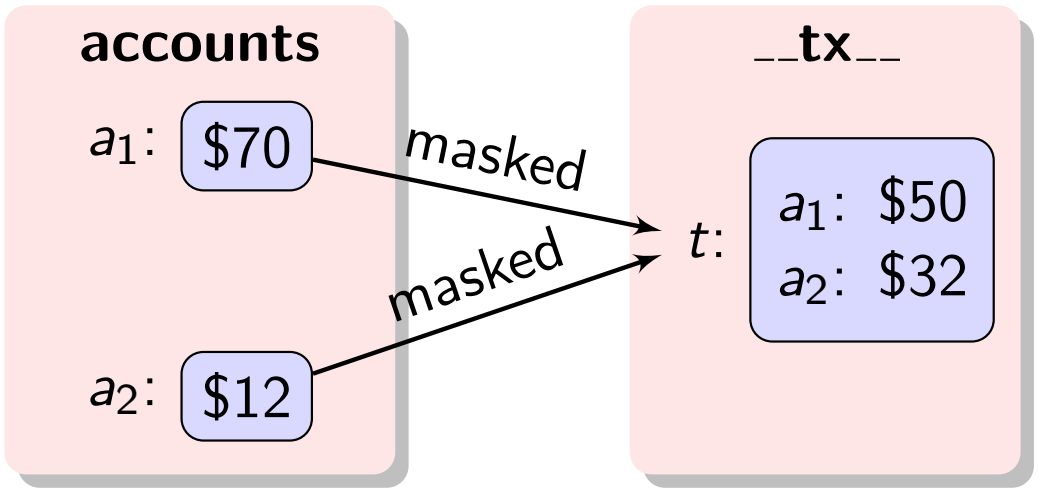
Transferring \$20 from a_1 to a_2:



Client-Side Transactions

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**

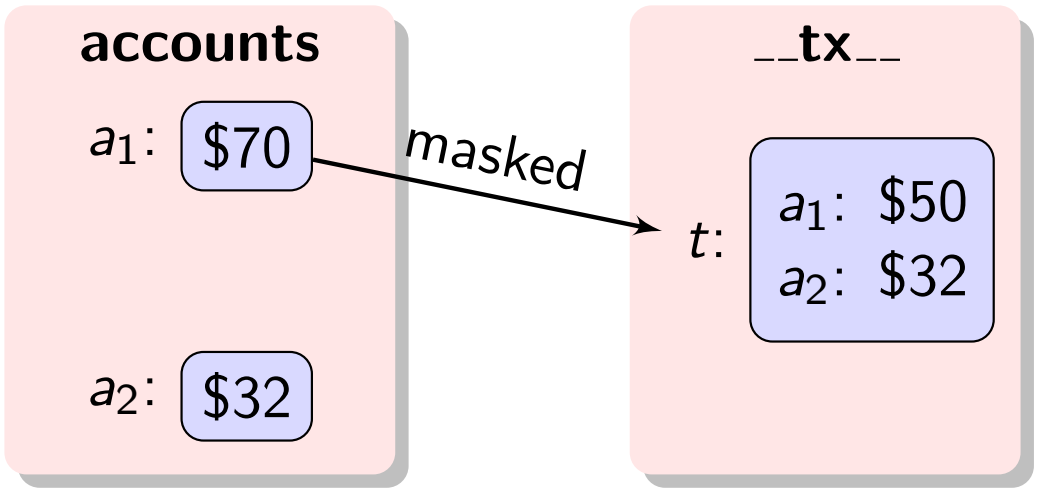
Transferring \$20 from a_1 to a_2:



Client-Side Transactions

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)

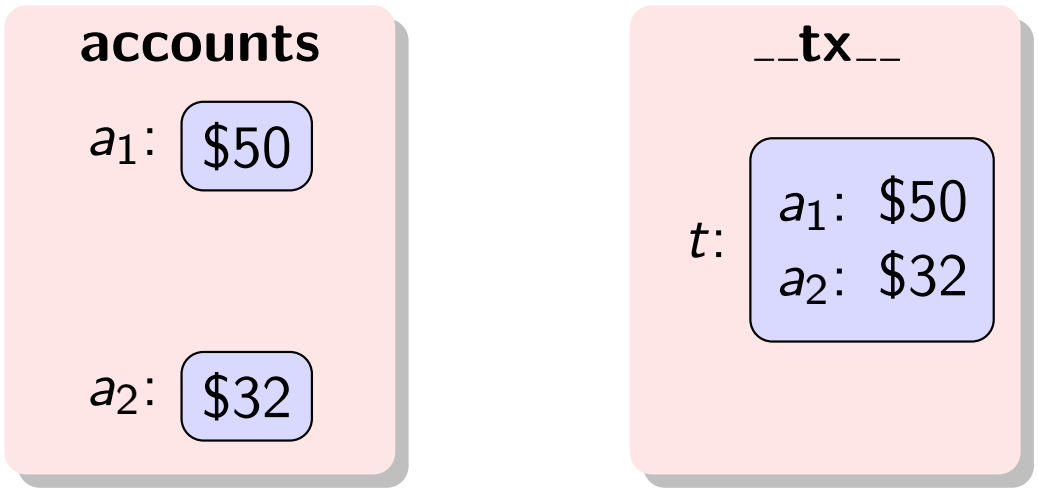
Transferring \$20 from a_1 to a_2:



Client-Side Transactions

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)

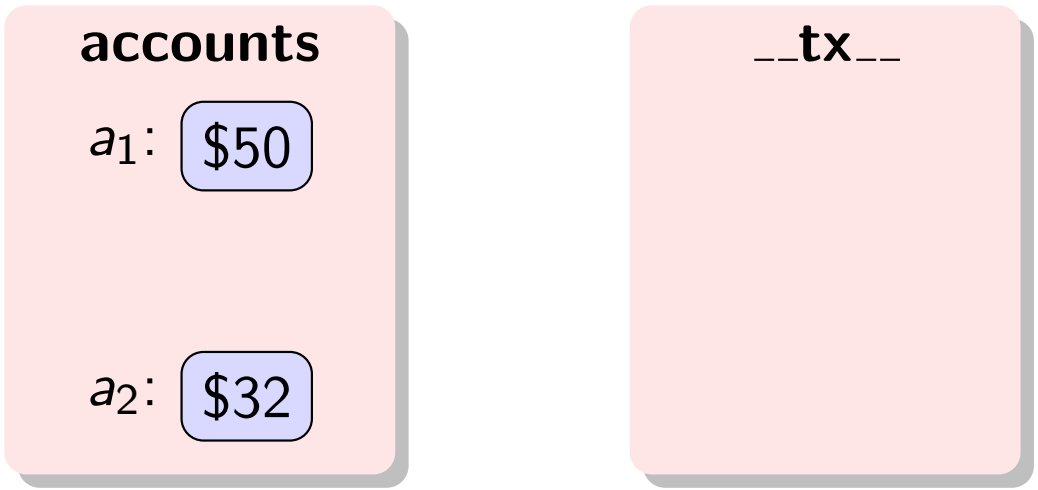
Transferring \$20 from a_1 to a_2:



Client-Side Transactions

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

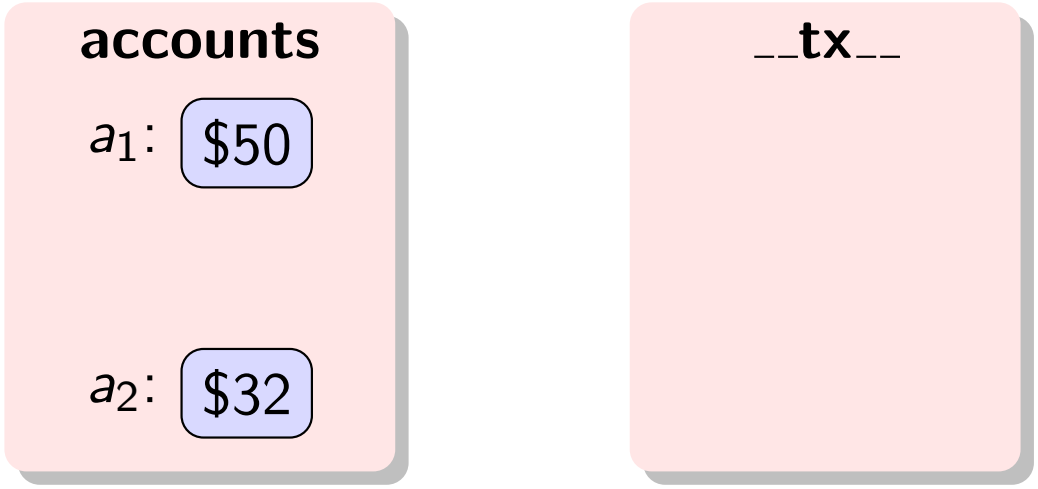
Transferring \$20 from a_1 to a_2:



Client-Side Transactions

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:

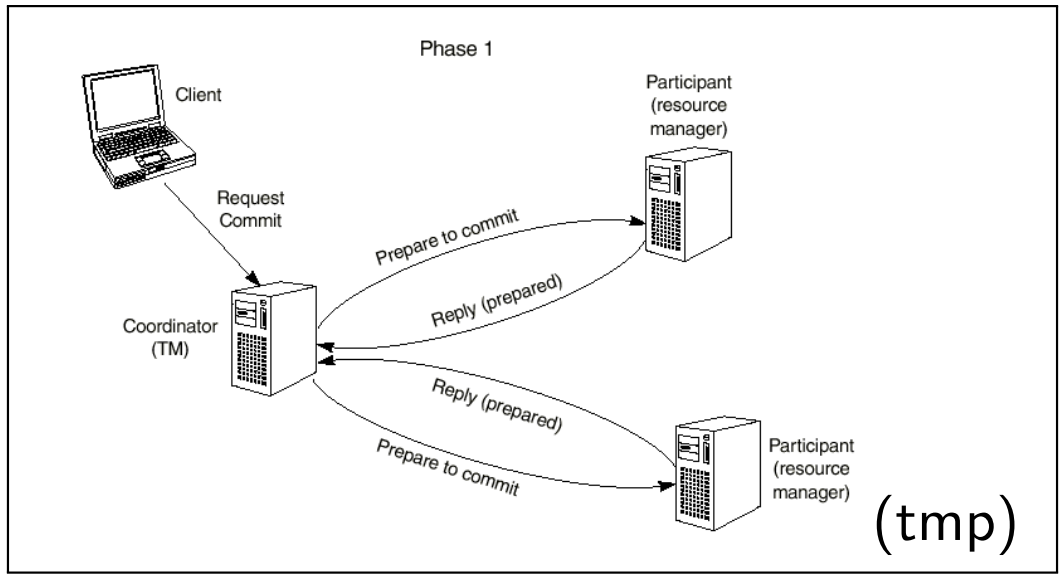


Options/Optimizations

- Server-side changes for cheap masking (step 1)
- Masters execute protocol on behalf on an application
- Pessimism

Two Phase Commit

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



Options/Optimizations

- App acts as transaction coordinator

Client-Side vs 2PC Comparison

Back-of-the-Envelope Performance

	Client-Side	Client-Side (server mods)	2PC	2PC (app coord)
log bytes	$3sn$	$2sn$	sn	sn
log writes	$2n+2$	$2n+2$	$2n$	$2n$
net bytes	$3sn$	$2sn$	$2sn$	sn
net RPCs	$2n+2$	$2n+2$	$2n-1$	$2n$

- s is the size of the objects
- n is number of objects and the number of participants (assuming all objects on different hosts)
- *net* numbers pretend logs are local to hosts

What's this hiding?

- Client-Side (both): depends on weak access control
- 2PC (app coord): one less serial log write in critical path
- Complexity

Relaxed Isolation

While the transaction commit is isolated, reads are not.

Example

Database invariant: $x + y = \$10$

T1	**T2**
	read(x) $\rightarrow$ \$6
read(x) $\rightarrow$ \$6	
	read(y) $\rightarrow$ \$4
	commit(\$3 $\rightarrow x$, \$7 $\rightarrow y$)
read(y) $\rightarrow$ \$7	
panic	

This is a general problem with optimistic concurrency control.

Relaxed Isolation

While the transaction commit is isolated, reads are not.

Example

Database invariant: $x + y = \$10$

T1	**T2**
	read(x) $\rightarrow$ \$6
read(x) $\rightarrow$ \$6	
	read(y) $\rightarrow$ \$4
	commit(\$3 $\rightarrow x$, \$7 $\rightarrow y$)
read(y) $\rightarrow$ \$7	
panic	

This is a general problem with optimistic concurrency control.

Options

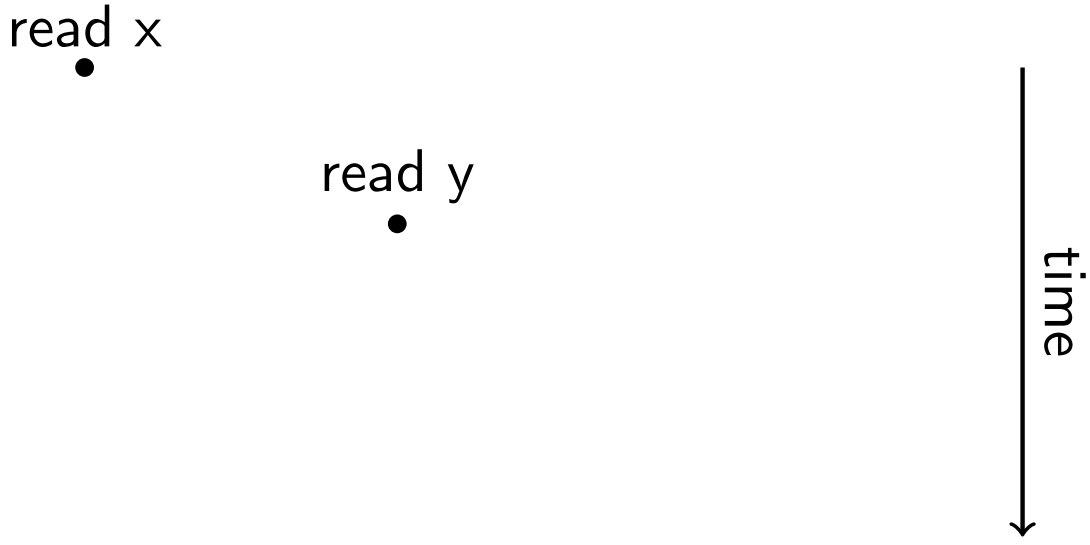
- Make the app deal with it
  - Must use caution at every exit path
- Snapshot the database on transaction start – too expensive
- Read set validation (early abort)

Read Set Validation

Example: read x, y, and z

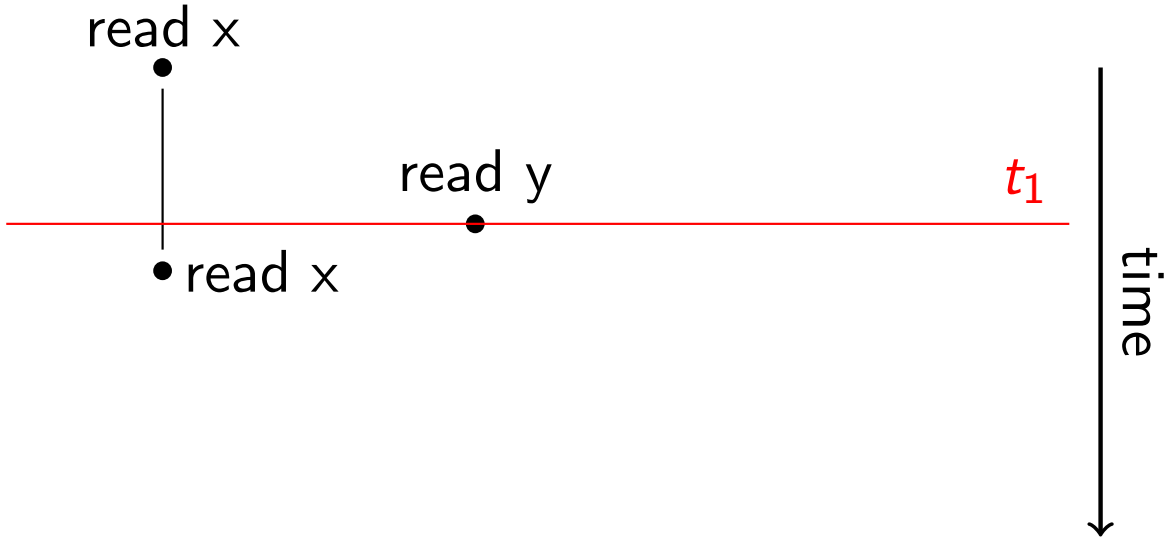
Read Set Validation

Example: read x, y, and z



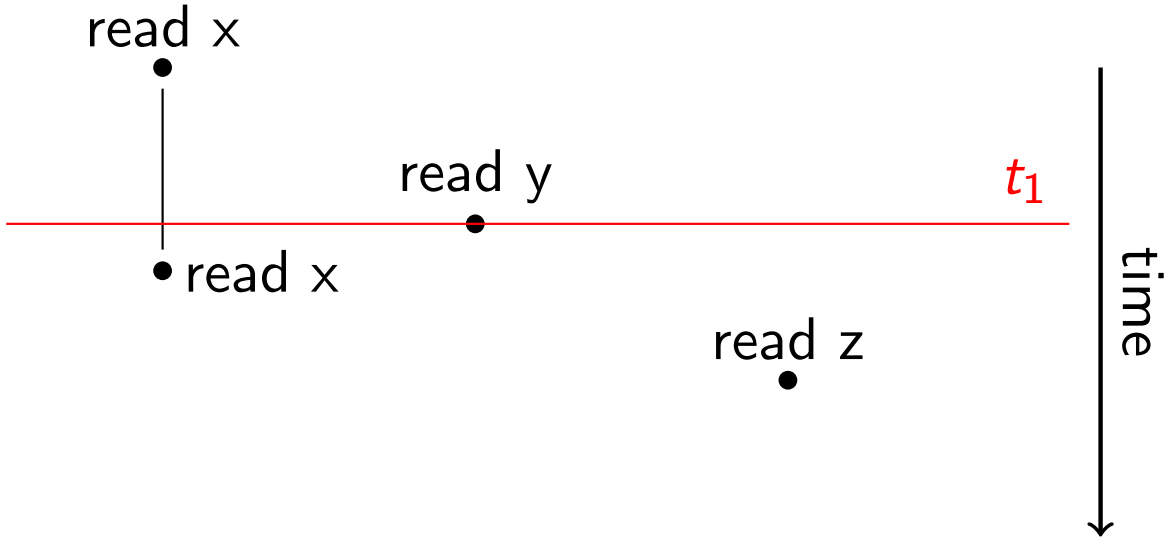
Read Set Validation

Example: read x, y, and z



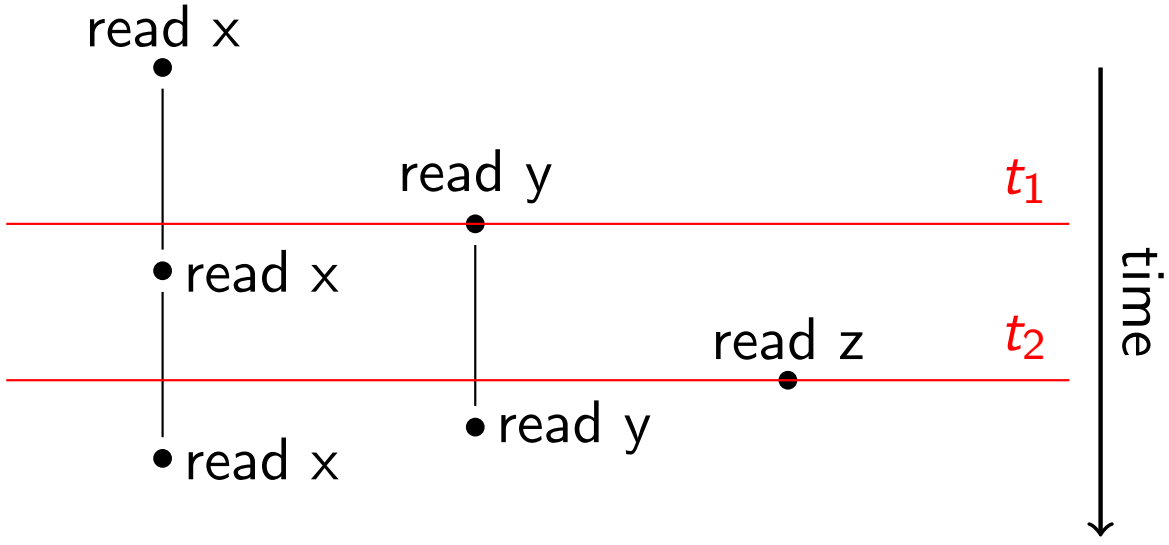
Read Set Validation

Example: read x, y, and z



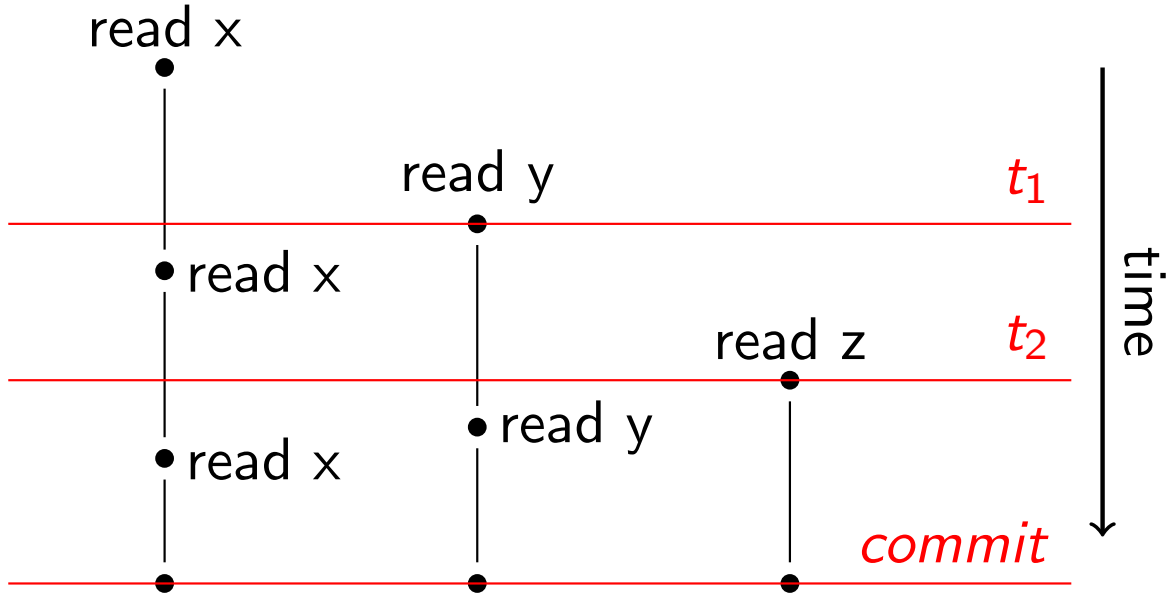
Read Set Validation

Example: read x, y, and z



Read Set Validation

Example: read x, y, and z



- Inefficient: $O(n^2)$ RPCs
- We can usually avoid validation using independent clocks
  - See me off-line for details
- Useful outside of transactions that modify the database

Conclusion

- Conditional API provides atomic ops for a single object
- Optimistic transactions for multiple objects
  - Optimized client-side approach about 2x slower than 2PC
- Read set validation for isolated reads
  - Can usually infer isolation from timestamps instead
- Remaining challenge: exposing these mechanisms to apps in a simple and powerful way

Questions/Comments

Some for the audience:

- Are conflicts as rare as we think?
- Do transactions belong as a first-class mechanism in RAMCloud?
- Client-side vs. 2PC?
- Do isolated reads matter?