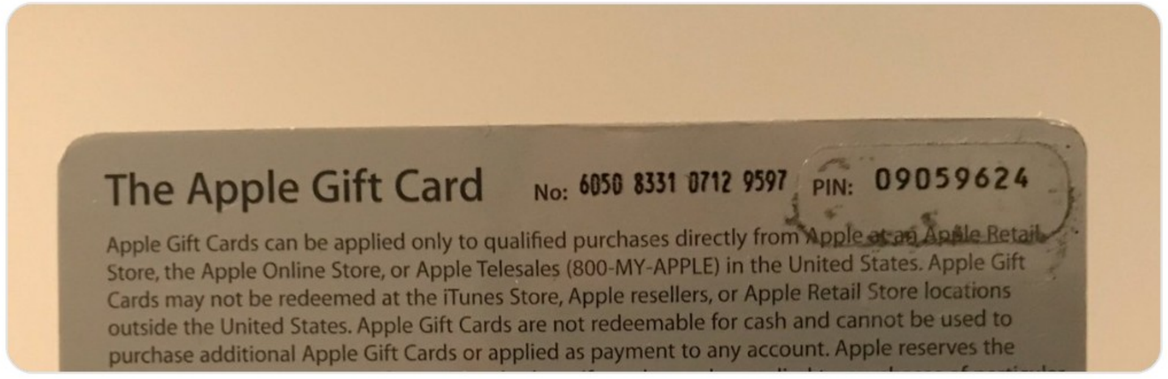
The cost of client-side distributed transactions

Denis Rystsov
@rystsov
derystso@microsoft.com
Microsoft / Cosmos DB

Do we need them?

Brad Fitzpatrick @bradfitz · 13 Jul 2017

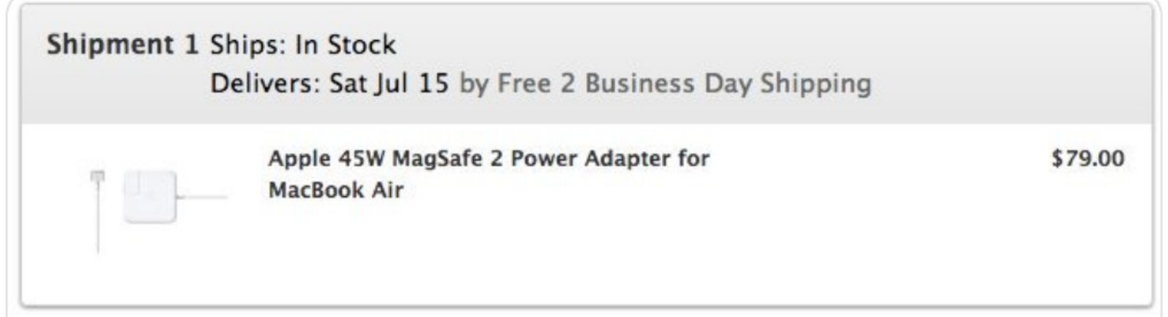
$34.95 balance on this Apple Gift Card I keep losing and forgetting to use for years. You're probably better at spending it than me. Enjoy.



6 80

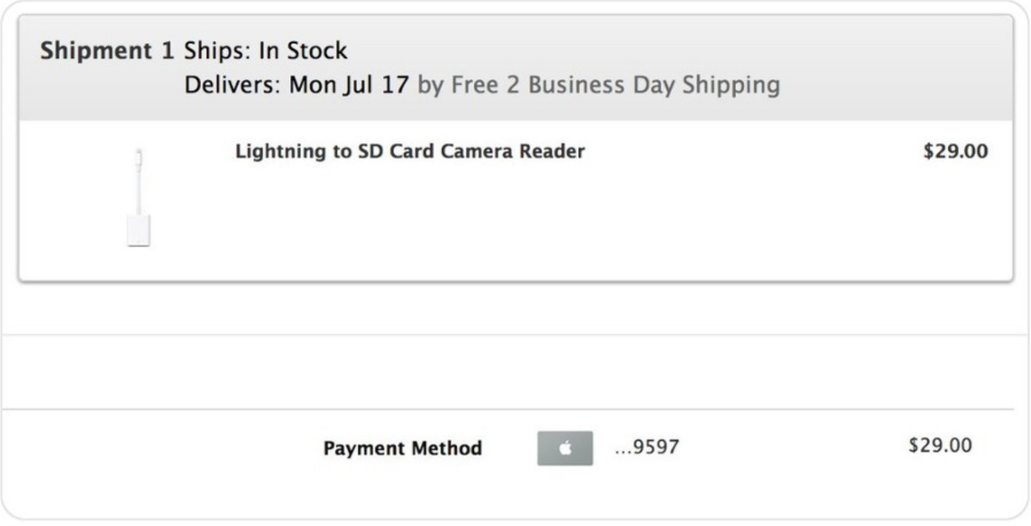
Harry Heymann @harryh · 13 Jul 2017

Thanks!



Aaron Parecki @aaronpk · 13 Jul 2017

hmm race condition? 🤔



1 8

Harry Heymann
@harryh

Follow

Replying to @aaronpk @bradfitz

A synchronization bug in Apple's gift card handling code? I would be shocked, just shocked to learn such a thing was possible!


HACKERS
12
UNITED ARTISTS PICTURES presents in IAIN SOFTLEY film "HACKERS" JONNY LEE MILLER ANGELINA JOLIE FISHER STEVENS and LORRAINE BRACCO music by SIMON BOSWELL edited by CHRISTOPHER BLUNDEN MARTIN WALSH

Homakov 21 мая 2015

Как я взломал Starbucks для безлимитного кофе

Это история о том, как я [illegible] неограниченное число денег на подарочные карты старбакса, тем са[illegible] бесплатный кофе, ну или украсть у них пару миллионов др[illegible]

How I hacked Starbucks for a life time coffee supply

Итак, не так давно мне [illegible] рбакса по $5 каждая.

BTC Stolen from Poloniex

March 04, 2014, 08:31:32 AM #1

All deposits, withdrawals, and markets are functioning normally. No further BTC will be deducted from anyone's balance.

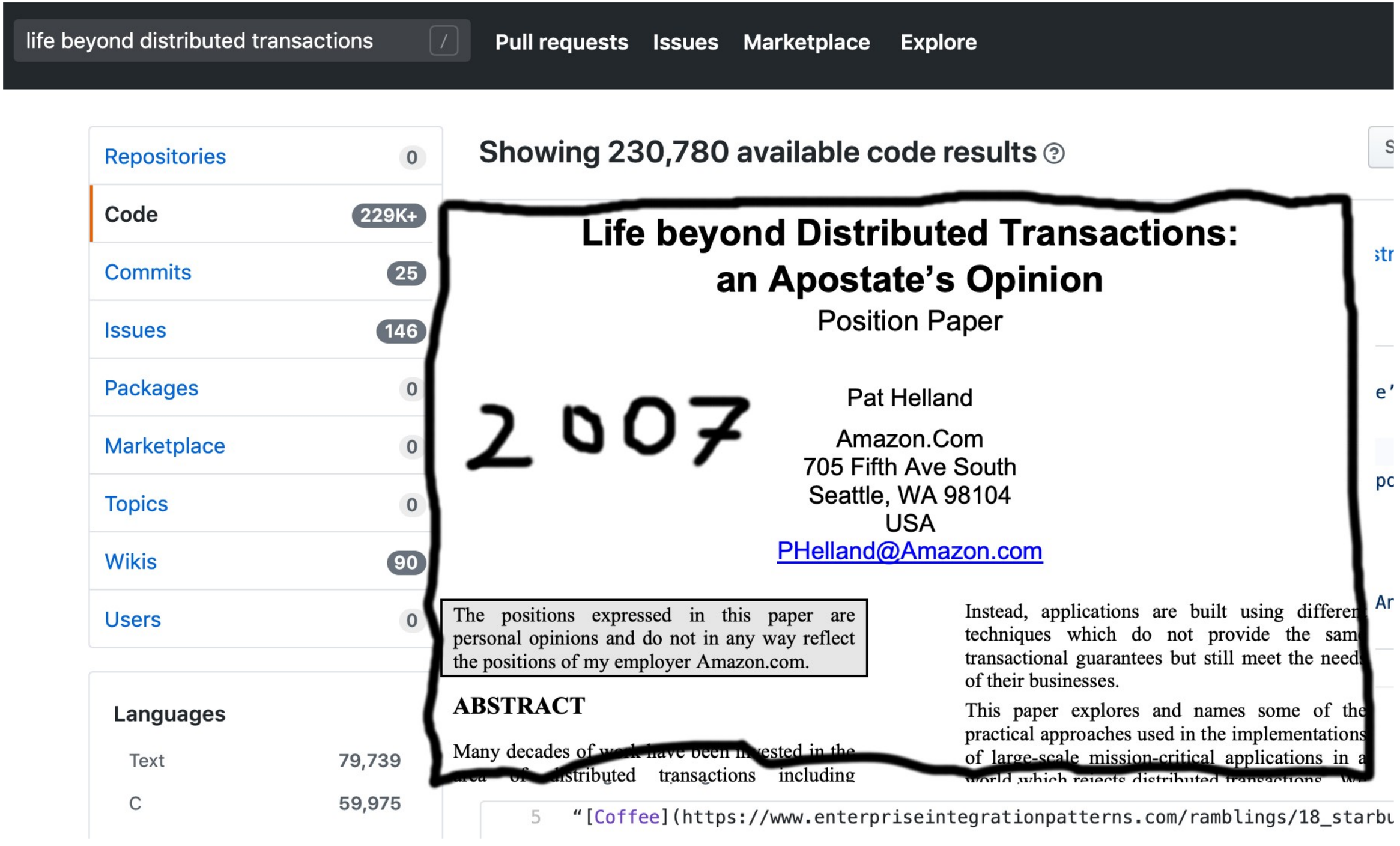
On March 4th, 2014, about 12.3% of the BTC on Poloniex was stolen.

How Did It Happen?

The hacker found a vulnerability in the code that takes withdrawals. Here's what happens when you place a withdrawal:

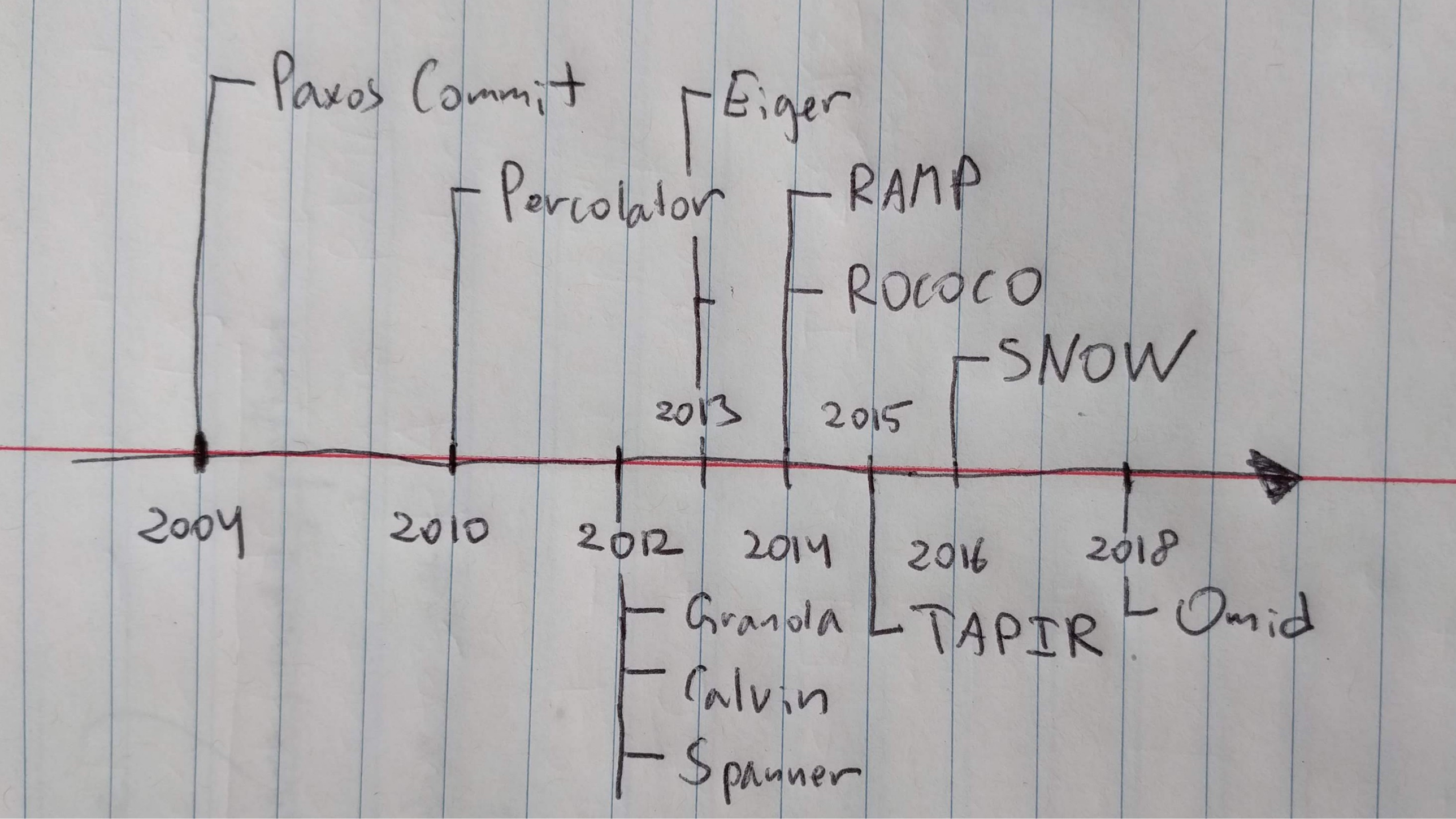
1. Input validation.
2. Your balance is checked to see if you have enough funds.
3. If you do, your balance is deducted.
4. The withdrawal is inserted into the database.
5. The confirmation email is sent.
6. After you confirm the withdrawal, the withdrawal daemon picks it up and processes the withdrawal.

The hacker discovered that if you place several withdrawals all in practically the same instant, they will get processed at more or less the same time. This will result in a negative balance, but valid insertions into the database, which then get picked up by the withdrawal daemon.


life beyond distributed transactions
/
Pull requests
Issues
Marketplace
Explore
Repositories 0
Code 229K+
Commits 25
Issues 146
Packages 0
Marketplace 0
Topics 0
Wikis 90
Users 0
Languages
Text 79,739
C 59,975
Showing 230,780 available code results
Life beyond Distributed Transactions:
an Apostate's Opinion
Position Paper
2007
Pat Helland
Amazon.Com
705 Fifth Ave South
Seattle, WA 98104
USA
PHelland@Amazon.com
The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer Amazon.com.
ABSTRACT
Many decades of work have been invested in the
distributed transactions including
Instead, applications are built using differen
techniques which do not provide the sam
transactional guarantees but still meet the need
of their businesses.
This paper explores and names some of the
practical approaches used in the implementations
of large-scale mission-critical applications in a
world which rejects distributed transactions
5 "[Coffee](https://www.enterpriseintegrationpatterns.com/ramblings/18_starbu

“When application developers attempt to use distributed transactions, the projects founder because the … costs and fragility make them impractical”

–Life beyond Distributed Transactions: an Apostate’s Opinion


Paxos Commit
Eiger
Percolator
RAMP
ROCOCO
SNOW
2013
2015
2004
2010
2012
2014
2016
2018
Granola
Calvin
Spanner
TAPIR
Omid

- What did fragility mean in 2000s?

- What did fragility mean in 2000s?
- Are distributed transactions still fragile in 2019?

- What did fragility mean in 2000s?
- Are distributed transactions still fragile in 2019?
- What's the cost of using distributed transactions now?

5 Answers

active oldest votes

30

The best way to distribute transactions over more than one database is: Don't.

Some people will point you to XA but XA (or Two Phase Commit) is a lie (or marketese).

...

share edit flag

answered May 20 '09 at 11:38

“if ... transaction coordinator becomes unavailable, then transactions and their associated data ... cannot be changed until the transaction outcome is resolved ... by external intervention”

CAP theorem

Availability or Strict Serializability

SNOW theorem

SNOW theorem

- **S**trict serializability

SNOW theorem

- **S**trict serializability
- **N**on blocking

SNOW theorem

- **S**trict serializability
- **N**on blocking
- return **O**ne value per read & do only **O**ne read

SNOW theorem

- **S**trict serializability
- **N**on blocking
- return **O**ne value per read & do only **O**ne read
- no conflicts with **W**rite transactions

- ~~What did fragility mean in 2000s (availability)?~~
- Do distributed transactions lead to unavailability in 2019?
- What's the cost of using distributed transactions now?

How to calculate cost?

How to calculate cost?

- Writing a distributed database isn’t a easy endeavor

How to calculate cost?

- Writing a distributed database isn’t a easy endeavor
- Deficiency in any one of a number of factors dooms it to failure

How to calculate cost?

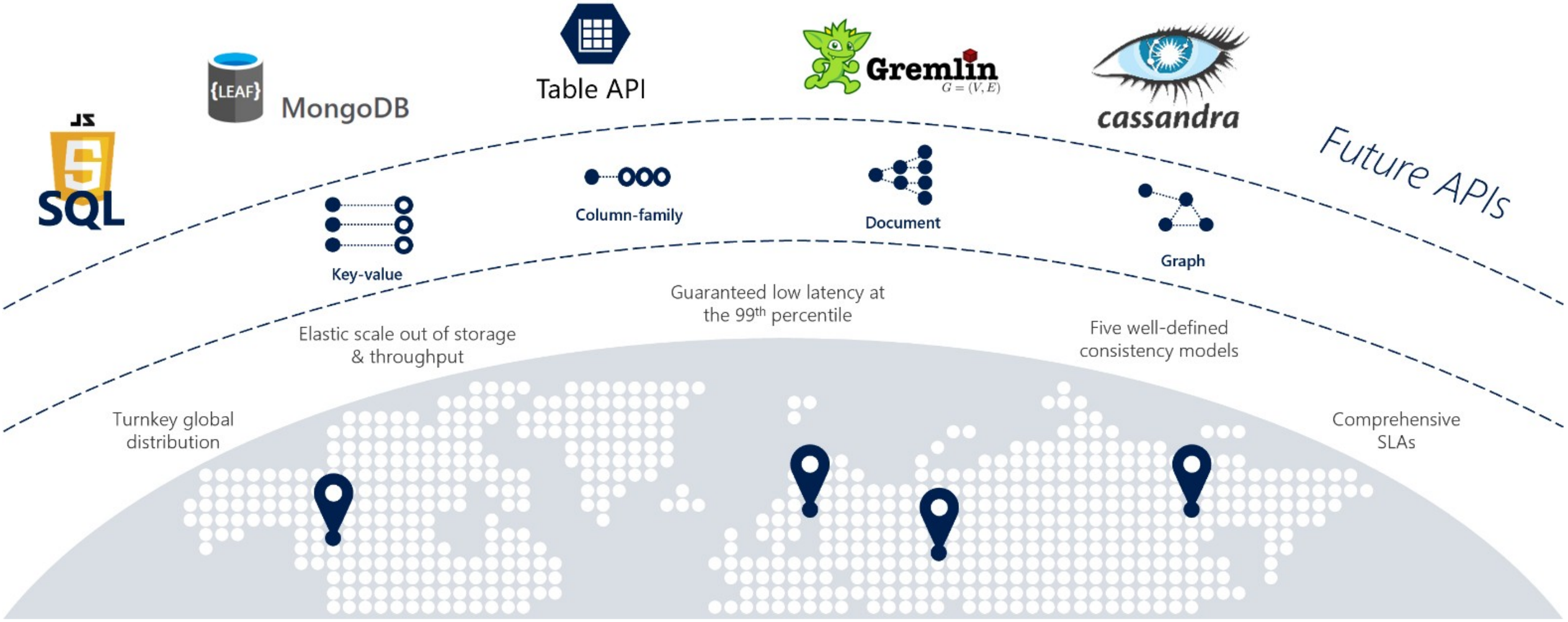
- Writing a distributed database isn’t a easy endeavor
- Deficiency in any one of a number of factors dooms it to failure
- Happy family are all alike; every unhappy family is unhappy in its own way

How to calculate cost?

- Writing a distributed database isn’t a easy endeavor
- Deficiency in any one of a number of factors dooms it to failure
- Happy family are all alike; every unhappy family is unhappy in its own way
- Calculate the cost with one DB, extrapolate on others

Azure Cosmos DB

Microsoft's globally distributed, massively scalable, multi-model database service



- The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (RU)

- The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (RU)
- RU provided for each request

- The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (RU)
- RU provided for each request
- Sum RUs to calculate cost of different protocols on the same workload

Why does the cost matter?

- $$$
- measure of work
- easy to measure

Workload

- A mixed workload with 80% reads and 20% writes

Workload

- A mixed workload with 80% reads and 20% writes
- Write tx transfers “money” between two accounts

Workload

- A mixed workload with 80% reads and 20% writes
- Write tx transfers “money” between two accounts
- Read tx reads balance from three accounts

Workload

- A mixed workload with 80% reads and 20% writes
- Write tx transfers “money” between two accounts
- Read tx reads balance from three accounts
- Fixed rate of collisions between keys of concurrent transaction

Workload

- A mixed workload with 80% reads and 20% writes
- Write tx transfers “money” between two accounts
- Read tx reads balance from three accounts
- Fixed rate of collisions between keys of concurrent transaction
- Retry until a fixed number of transactions are successfully executed

Workload

- A mixed workload with 80% reads and 20% writes
- Write tx transfers “money” between two accounts
- Read tx reads balance from three accounts
- Fixed rate of collisions between keys of concurrent transaction
- Retry until a fixed number of transactions are successfully executed
- Normalize total the cost by the number of successfully executed transactions

Granola	Janus	Sagas
RAMP	Calvin	Cure
Paxos Commit	TAPIR	COPS
Eiger	2PC	Occult
ROCOCO	Percolator	MDCC
Spanner	Omid	Walter

Selection criteria

- familiar model (ACID)

Selection criteria

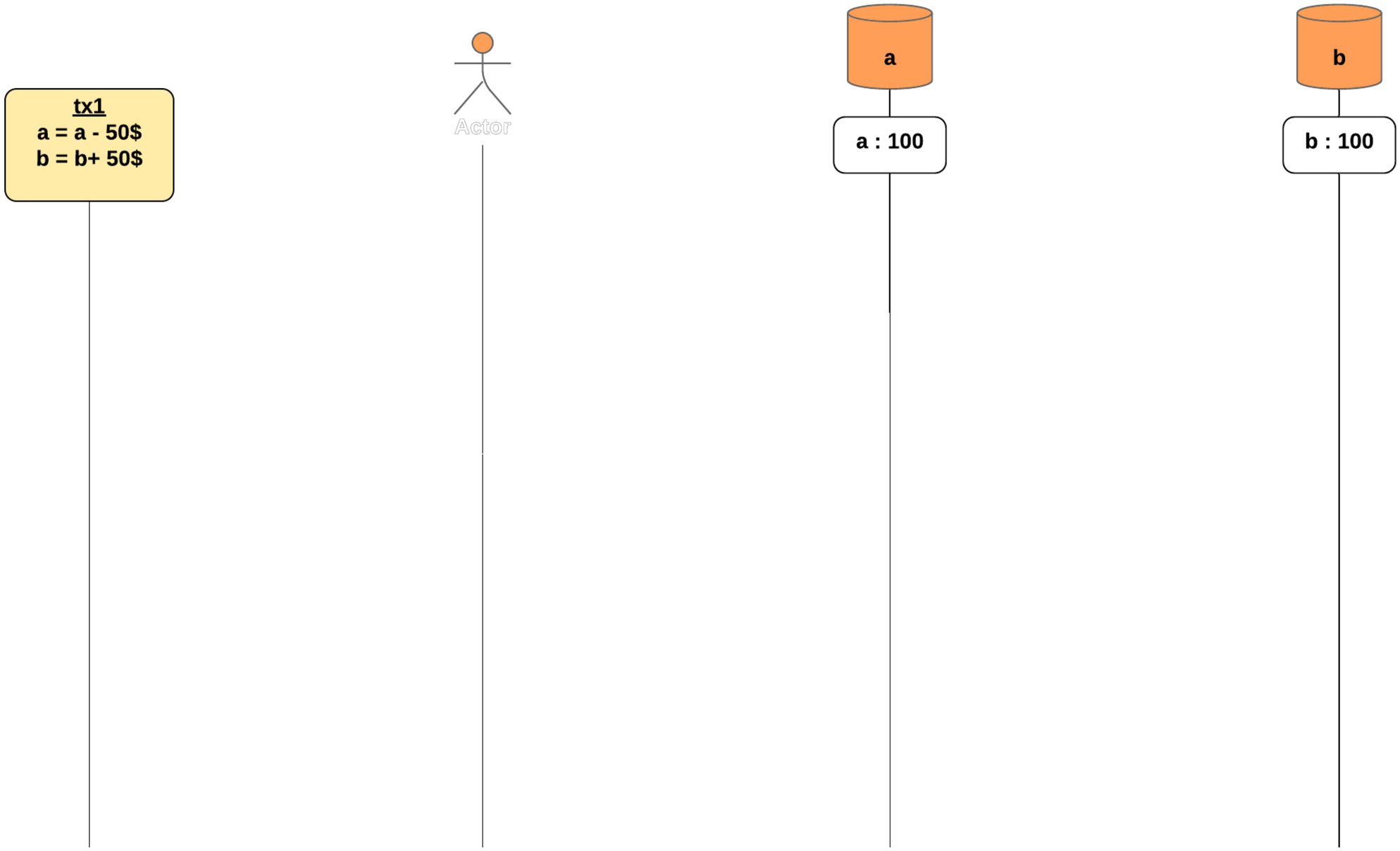
- familiar model (ACID)
- can be used with existing storages

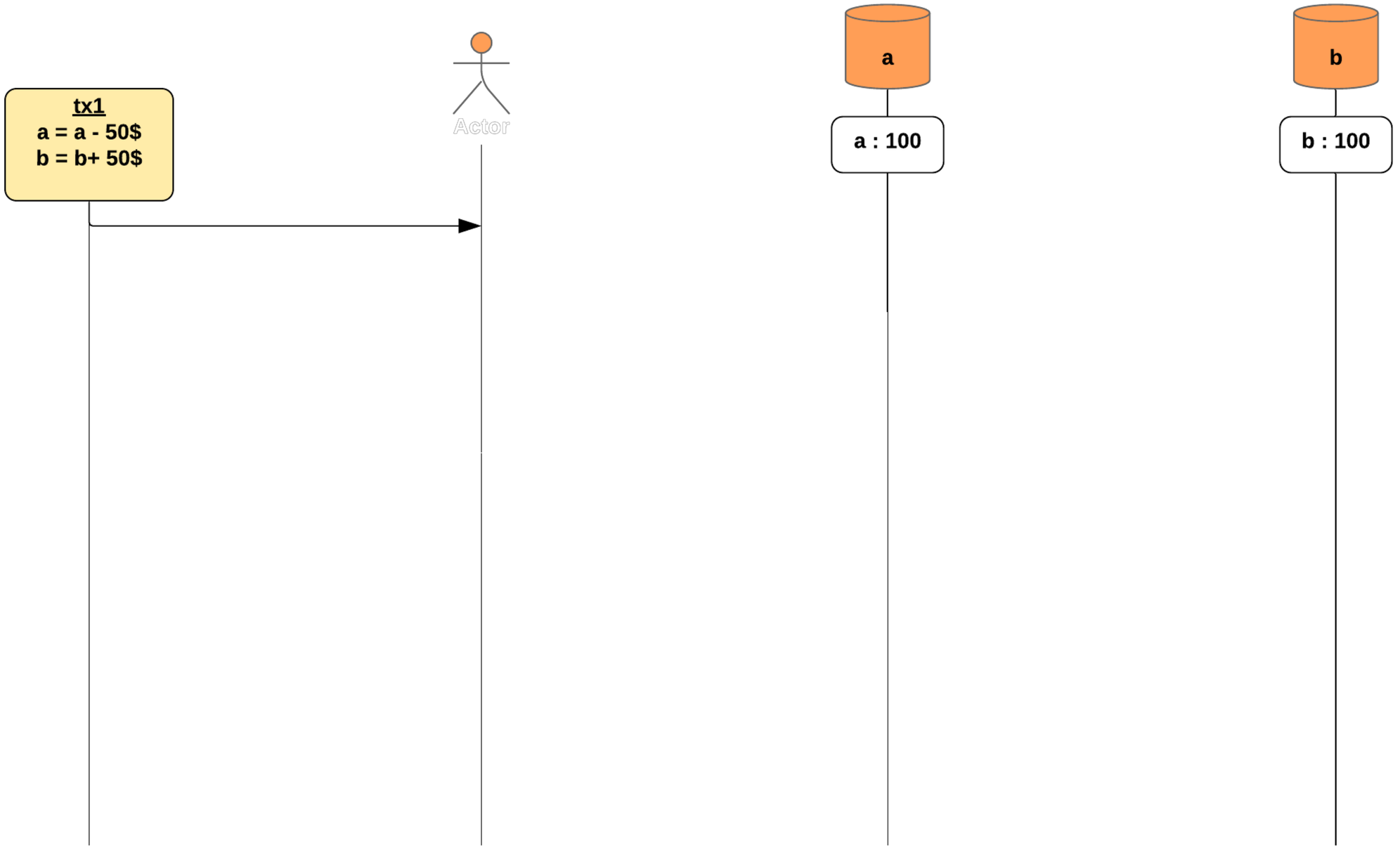
Selection criteria

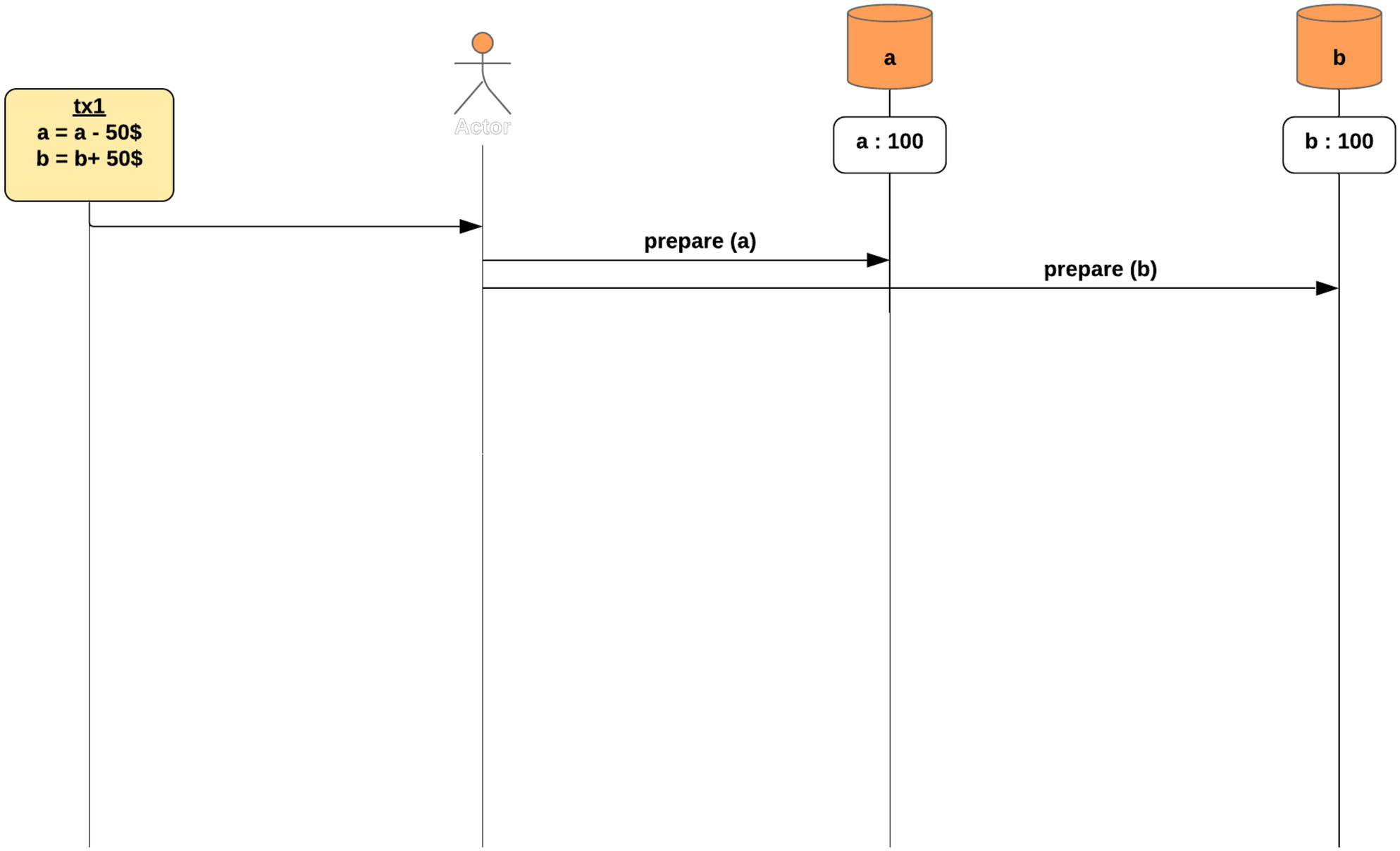
- familiar model (ACID)
- can be used with existing storages
- doesn’t require specific hardware like atomic clocks

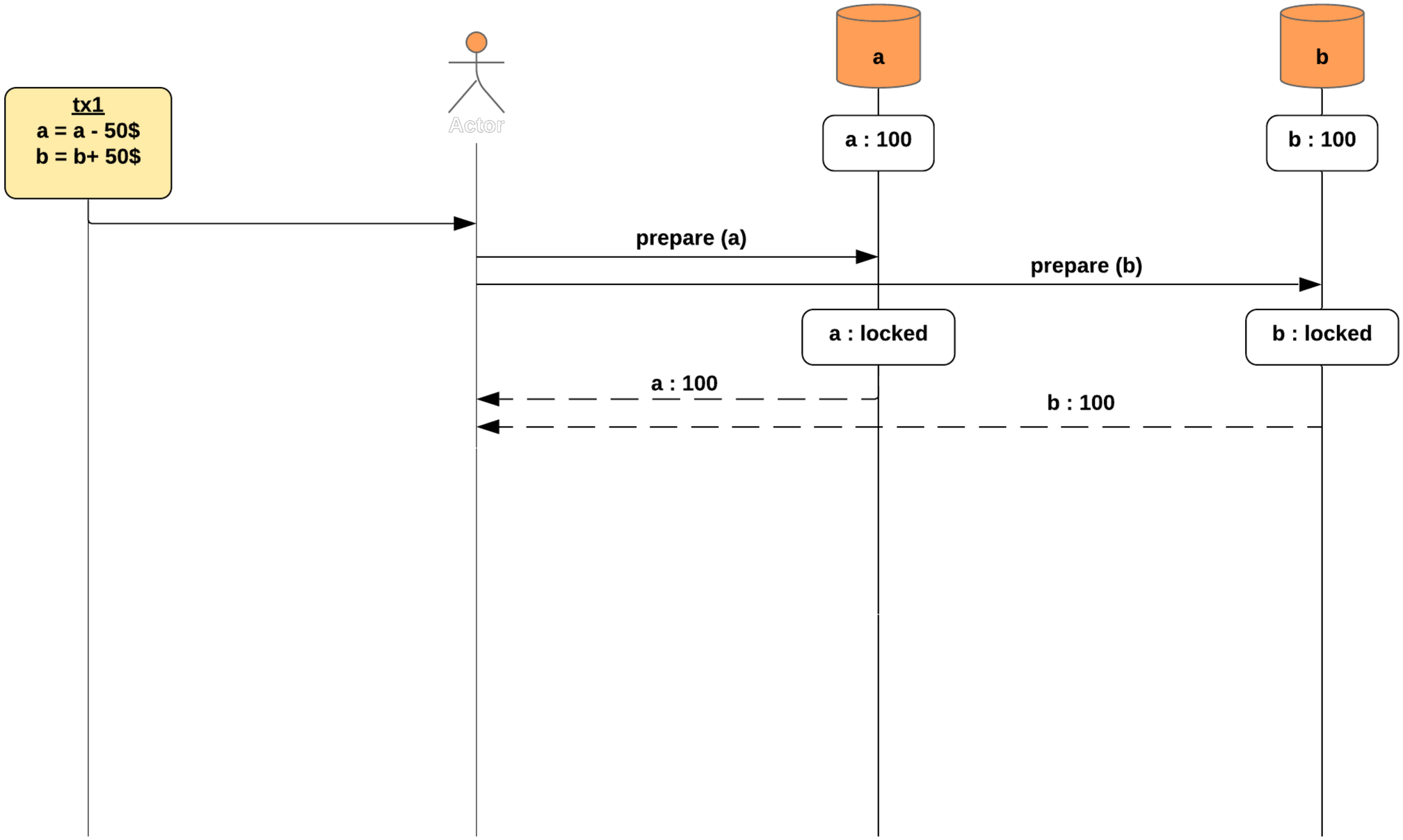
Serializability	Snapshot Isolation	RA (Read Committed+)
2 phase commit (2PC)	Percolator	RAMP
Paxos Commit (PC)		
Eiger		
Granola		

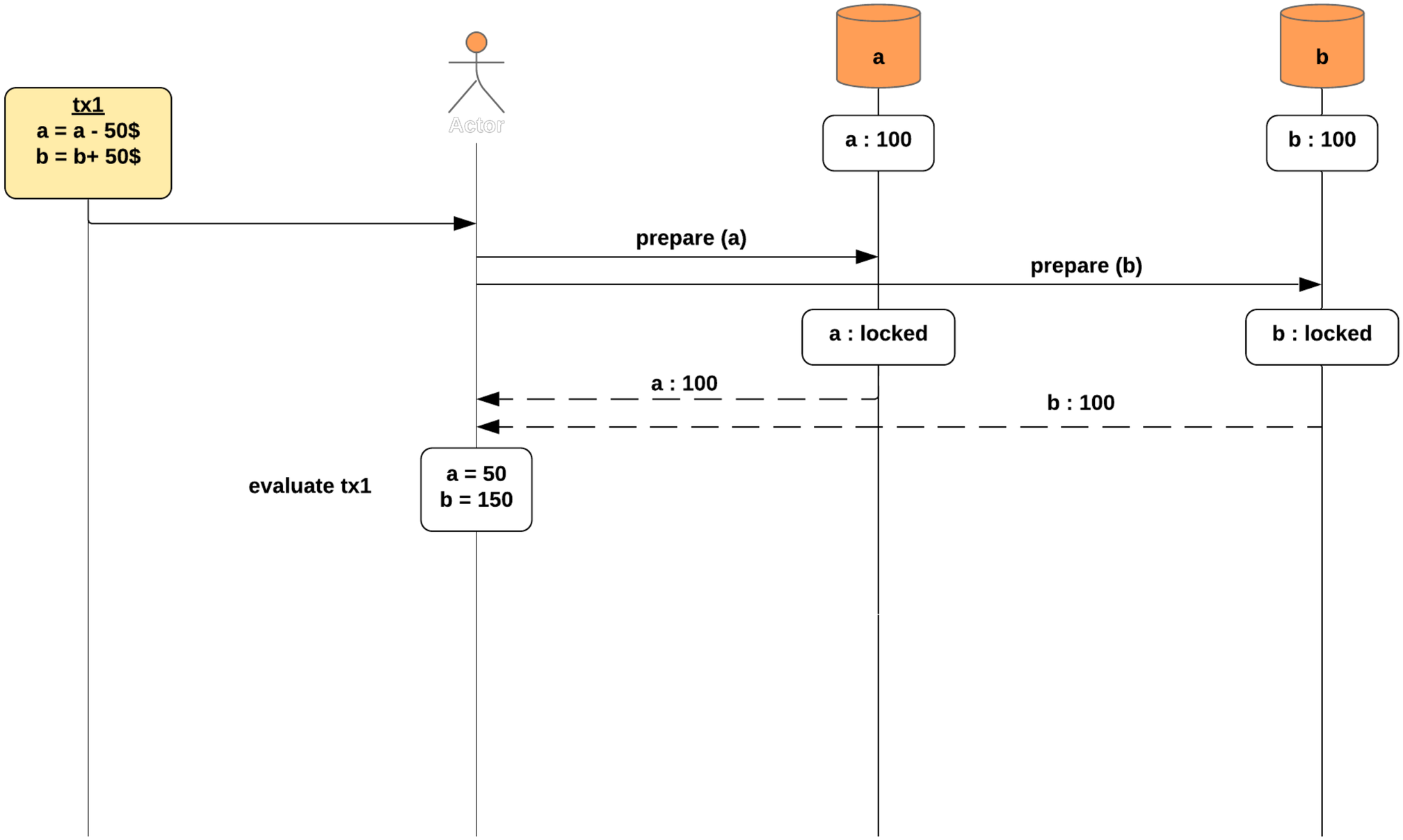
Two-Phase Commit

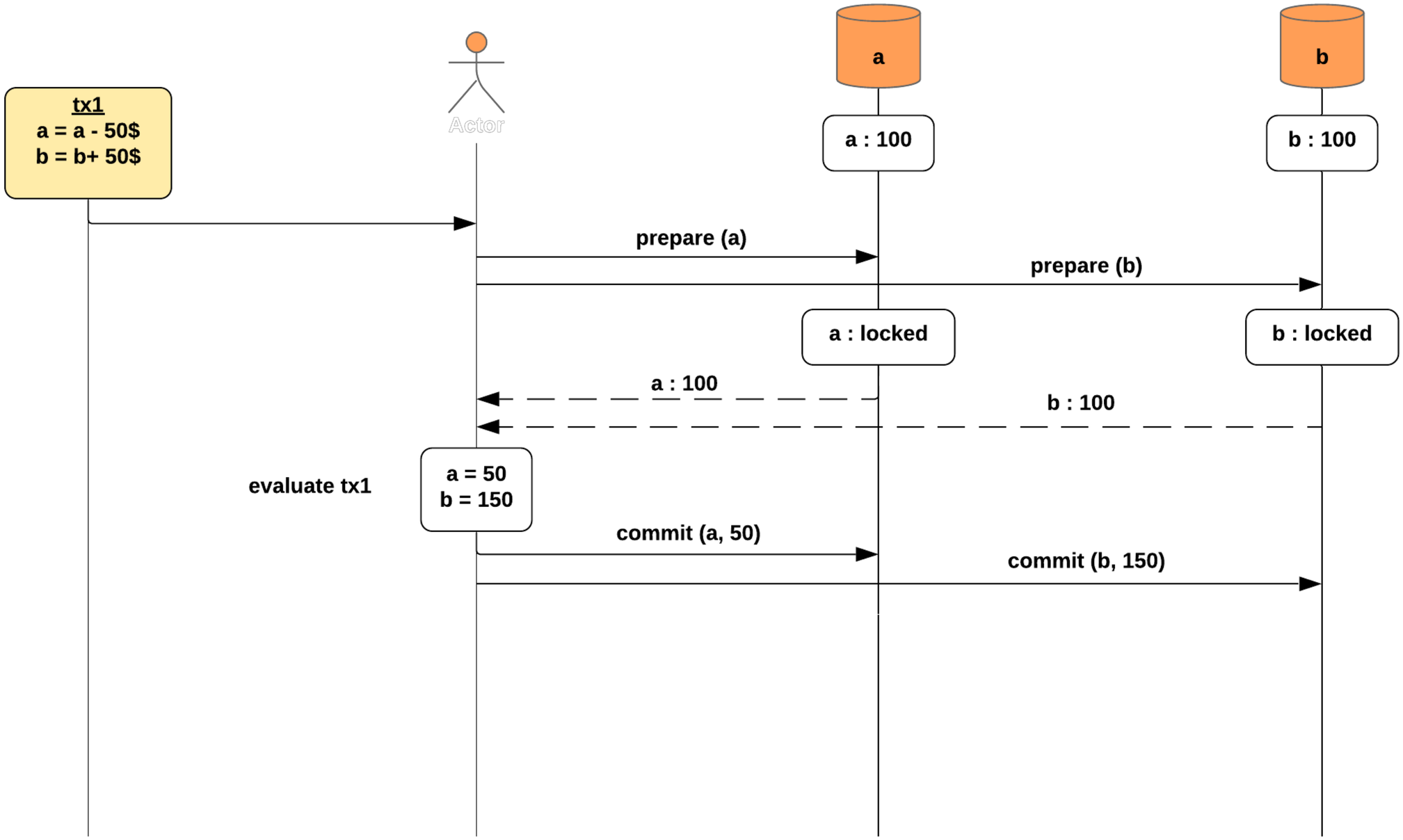

tx1
a = a - 50$
b = b+ 50$
Actor
a
a : 100
b
b : 100

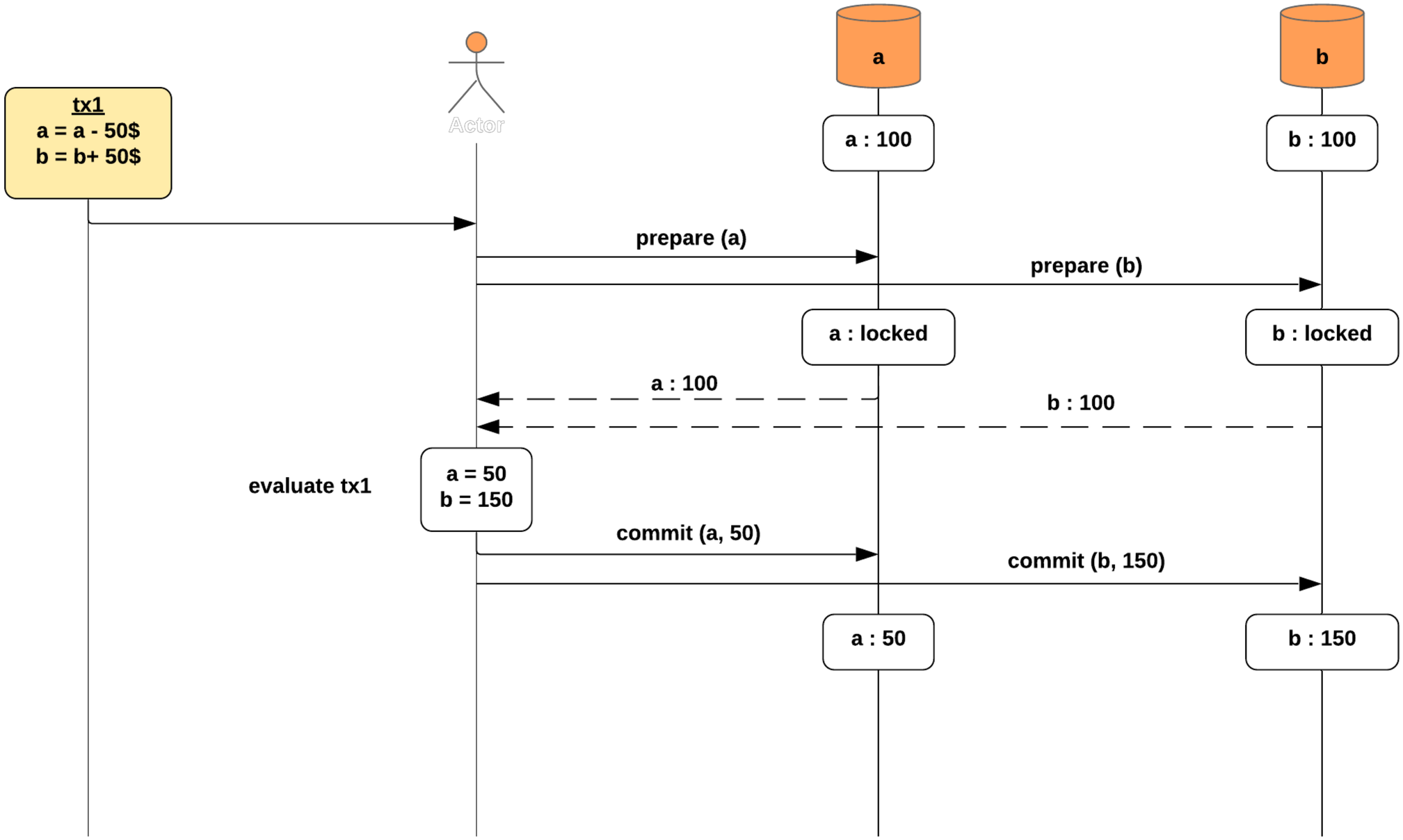

tx1
a = a - 50$
b = b+ 50$
Actor
a
a : 100
b
b : 100

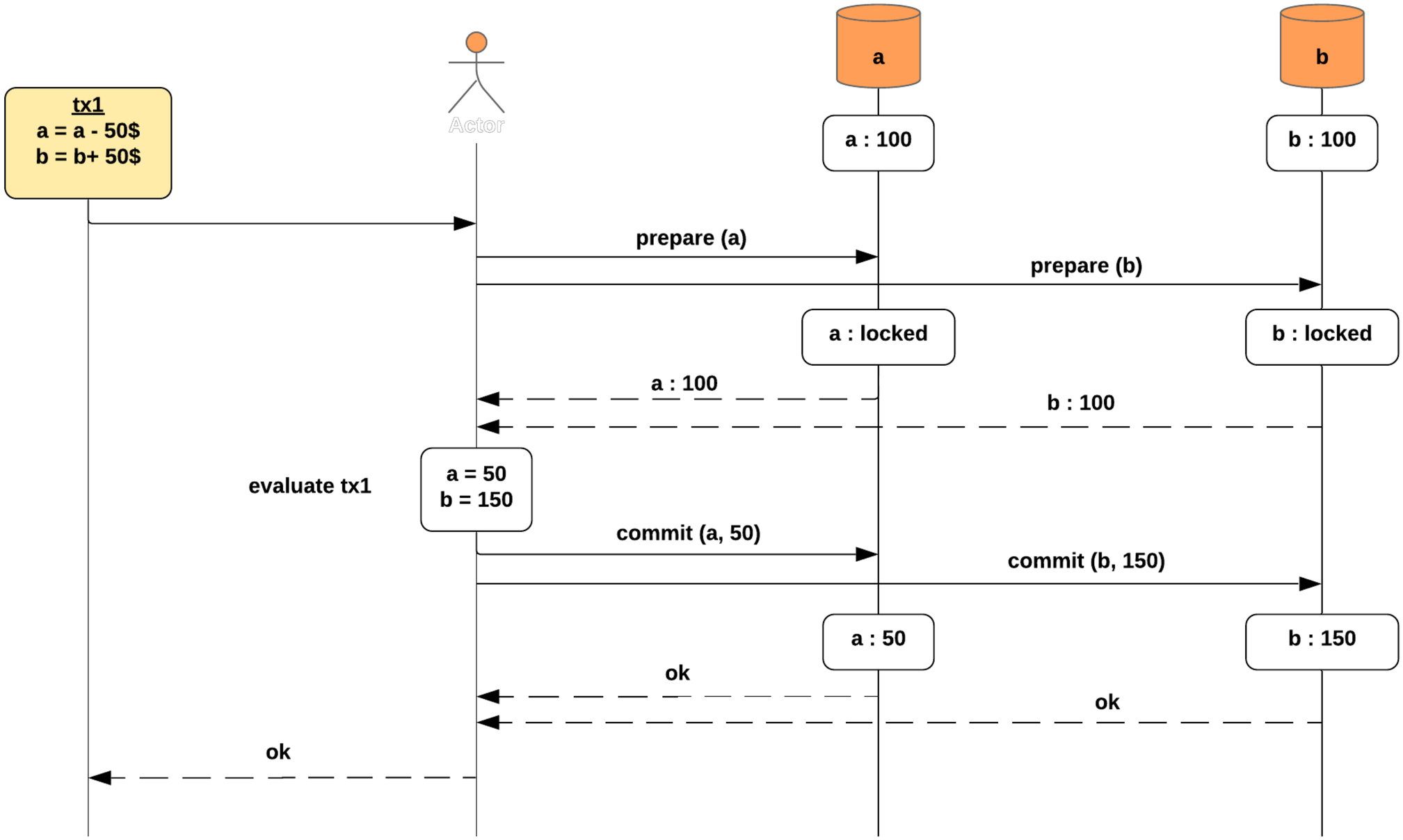

tx1
a = a - 50$
b = b+ 50$
Actor
a
b
a : 100
b : 100
prepare (a)
prepare (b)


tx1
a = a - 50$
b = b+ 50$
Actor
a
b
a : 100
b : 100
prepare (a)
prepare (b)
a : locked
b : locked
a : 100
b : 100

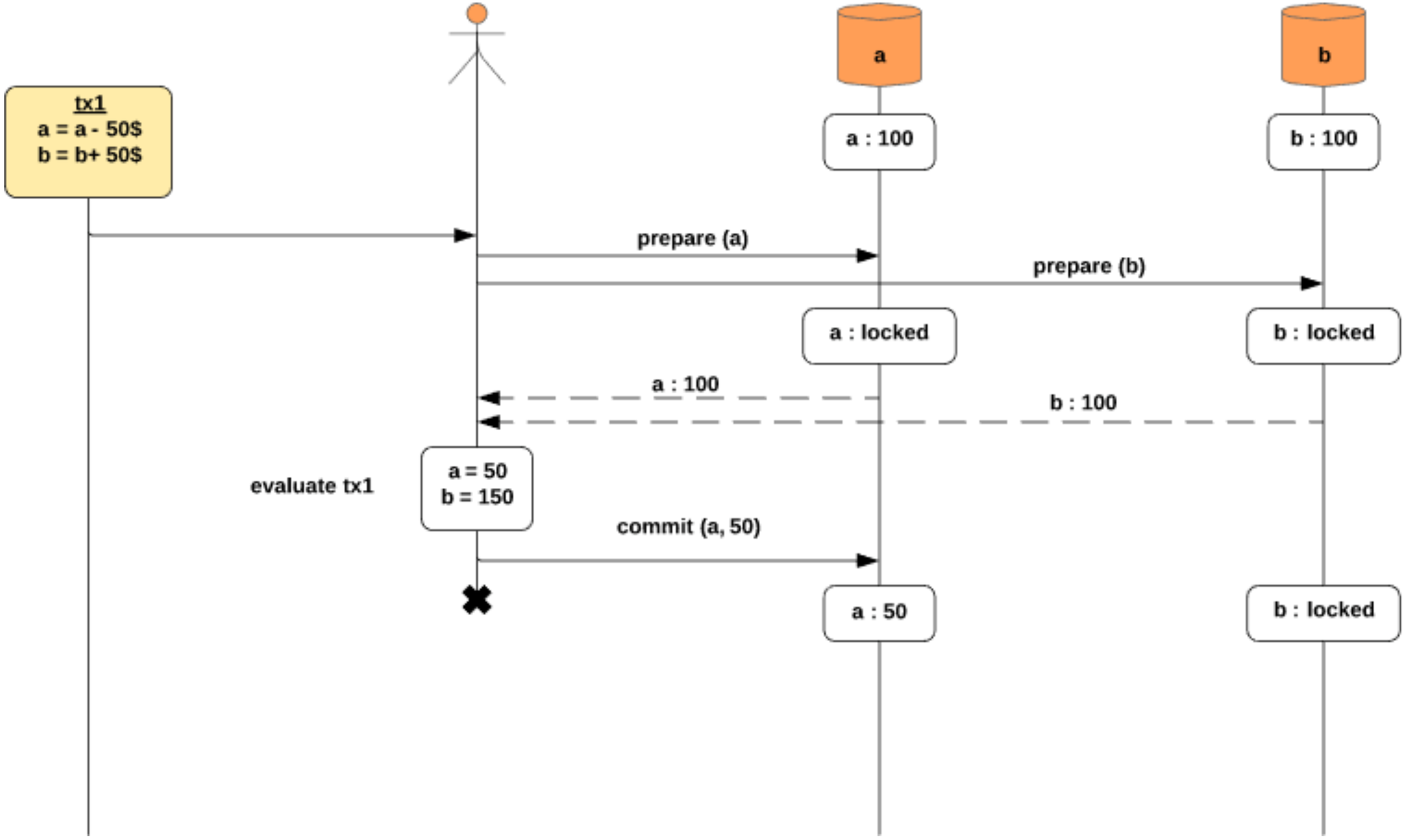

tx1
a = a - 50$
b = b+ 50$
Actor
a
b
a : 100
b : 100
prepare (a)
prepare (b)
a : locked
b : locked
a : 100
b : 100
evaluate tx1
a = 50
b = 150

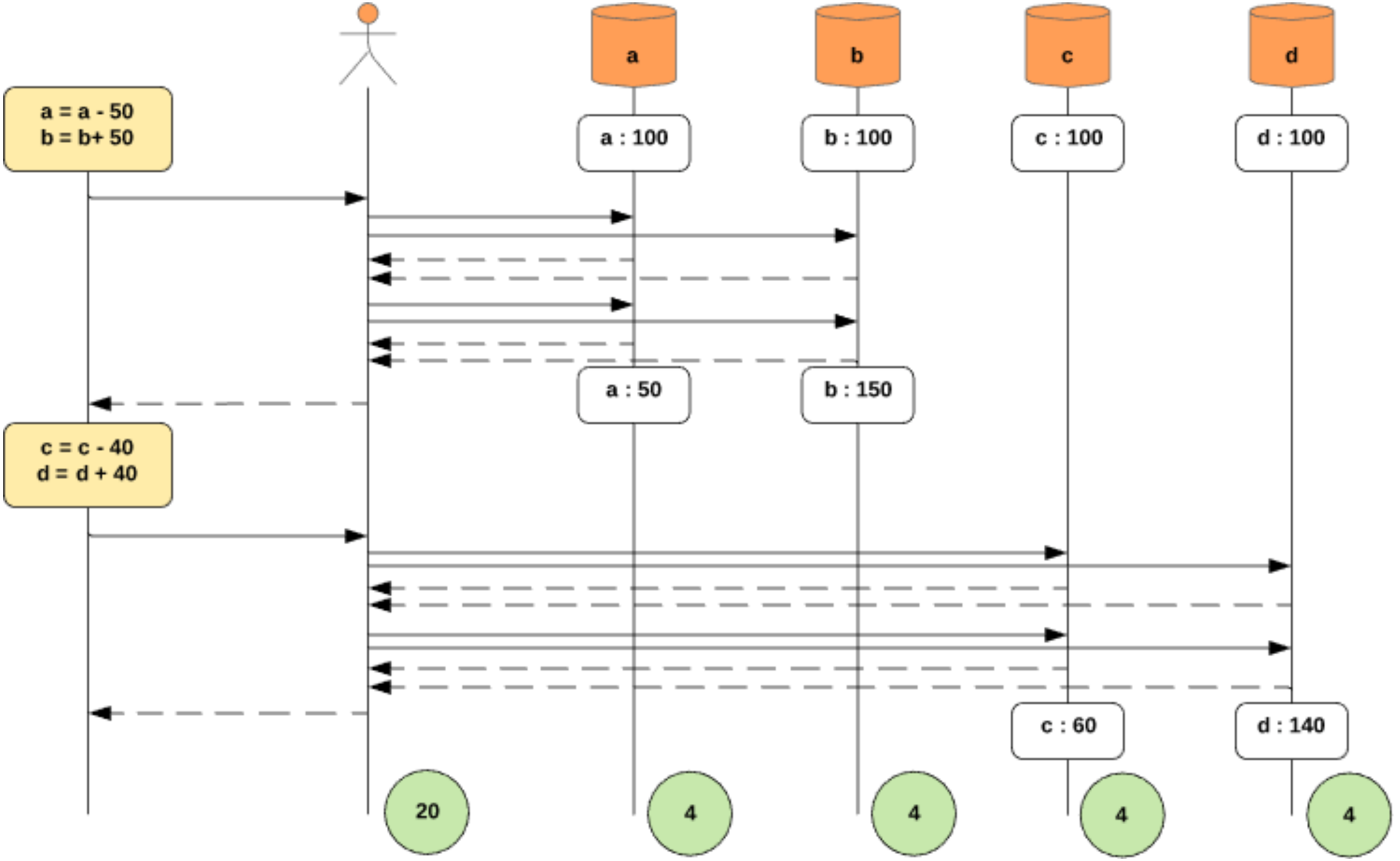

tx1
a = a - 50$
b = b+ 50$
Actor
a
b
a : 100
b : 100
prepare (a)
prepare (b)
a : locked
b : locked
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (a, 50)
commit (b, 150)


tx1
a = a - 50$
b = b+ 50$
Actor
a
b
a : 100
b : 100
prepare (a)
prepare (b)
a : locked
b : locked
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (a, 50)
commit (b, 150)
a : 50
b : 150


tx1
a = a - 50$
b = b+ 50$
Actor
a
b
a : 100
b : 100
prepare (a)
prepare (b)
a : locked
b : locked
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (a, 50)
commit (b, 150)
a : 50
b : 150
ok
ok
ok

Baseline	9 RU
2PC	14 RU
2PC	55%


tx1
a = a - 50$
b = b+ 50$
a
b
a : 100
b : 100
prepare (a)
prepare (b)
a : locked
b : locked
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (a, 50)
a : 50
b : locked


a
b
c
d
a = a - 50
b = b+ 50
a : 100
b : 100
c : 100
d : 100
a : 50
b : 150
c = c - 40
d = d + 40
c : 60
d : 140
20
4
4
4
4

2PC Problems

- Availability
- Scalability

2PC

- If a coordinator loses its state the databases will be blocked and require a DBA intervention

2PC + Paxos

- If a coordinator loses its state the databases will be blocked and require a DBA intervention
- A consensus (replication) protocol may help with state reliability

2PC + Paxos = Paxos Commit

- If a coordinator loses its state the databases will be blocked and require a DBA intervention
- A consensus (replication) protocol may help with state reliability
- A combination of Paxos and 2PC allows to have multiple solves availability problem and allow multiple coordinators coexist and distribute load

Paxos Commit

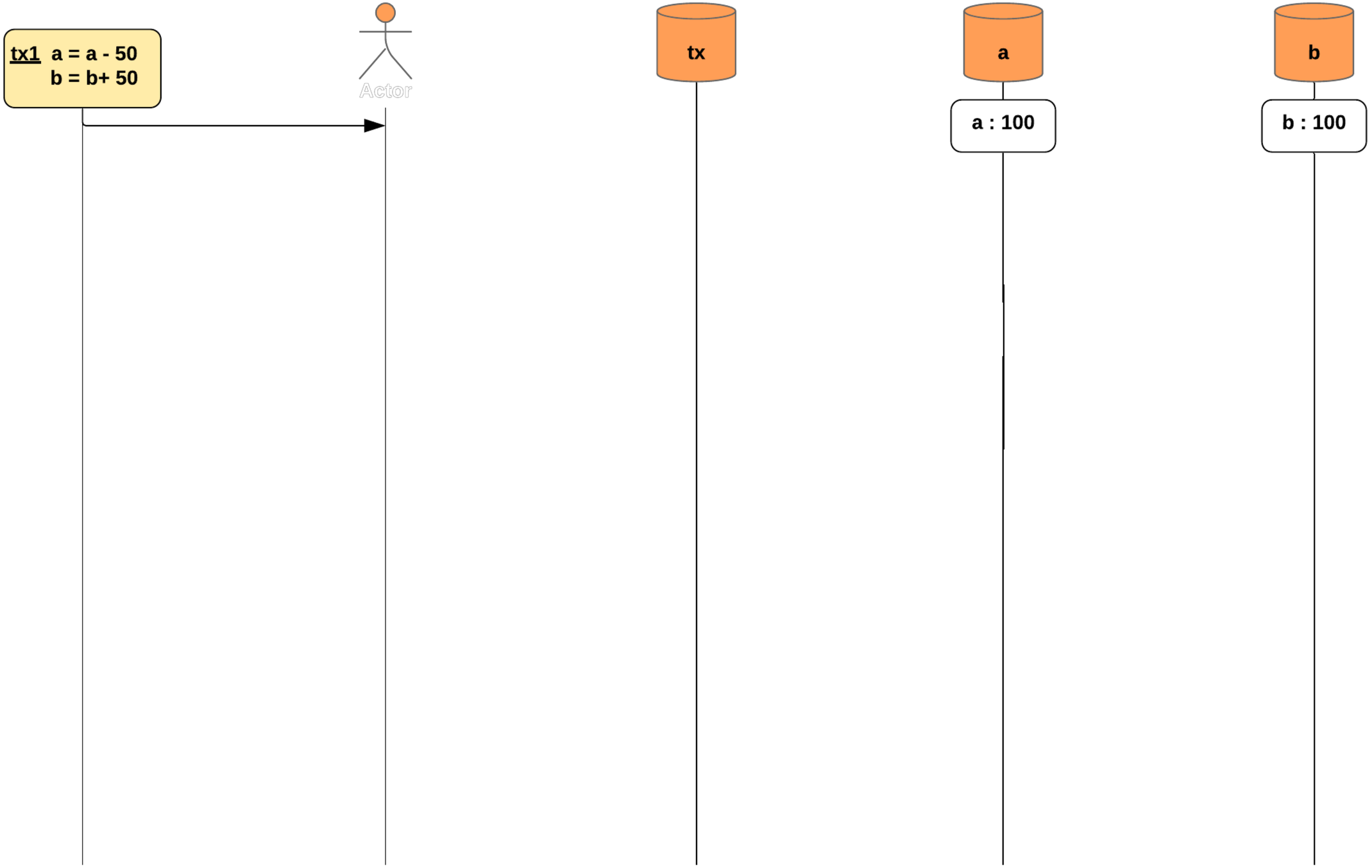
Consensus on Transaction Commit

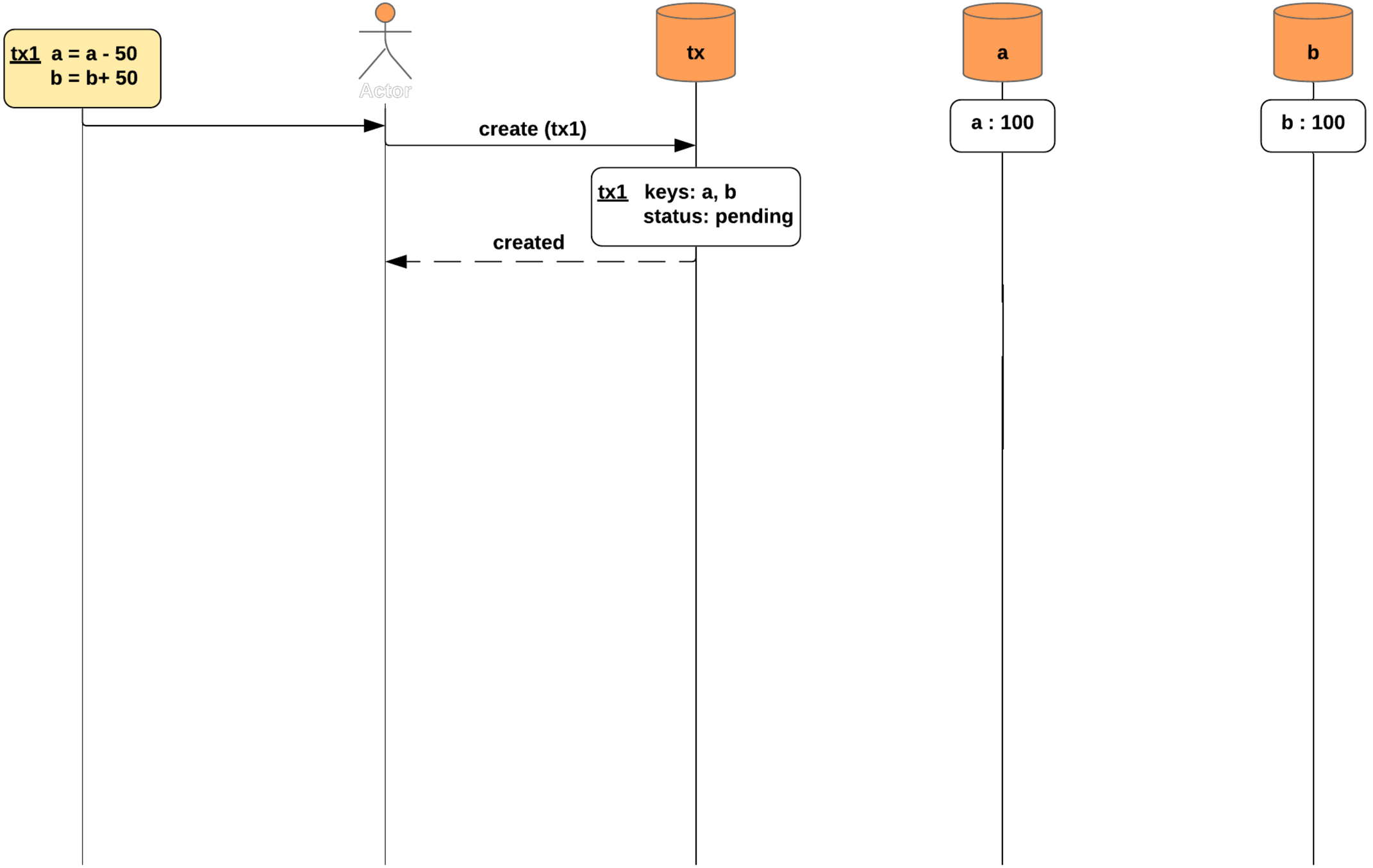
Jim Gray and Leslie Lamport
Microsoft Research

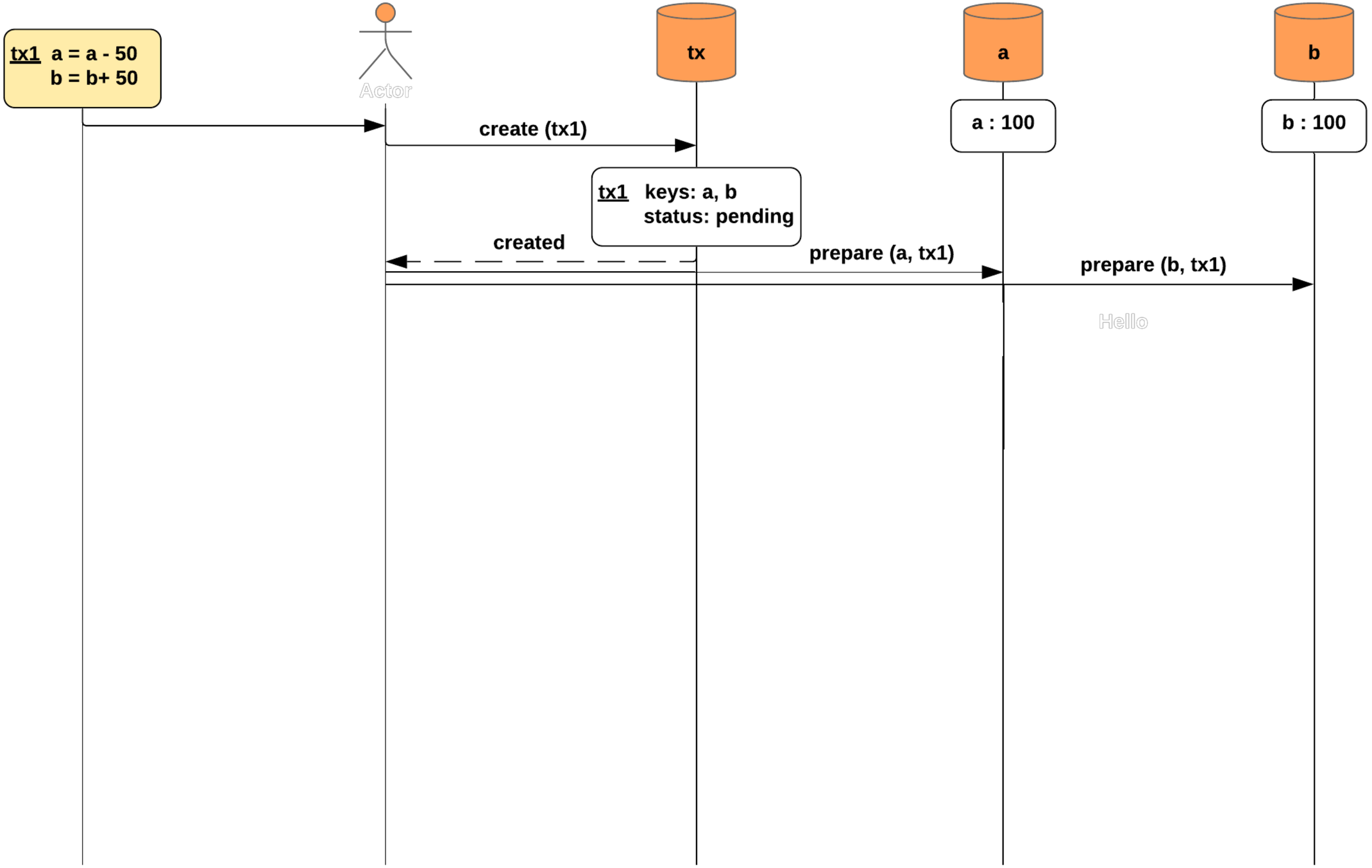
1 January 2004
revised 19 April 2004, 8 September 2005, 5 July 2017

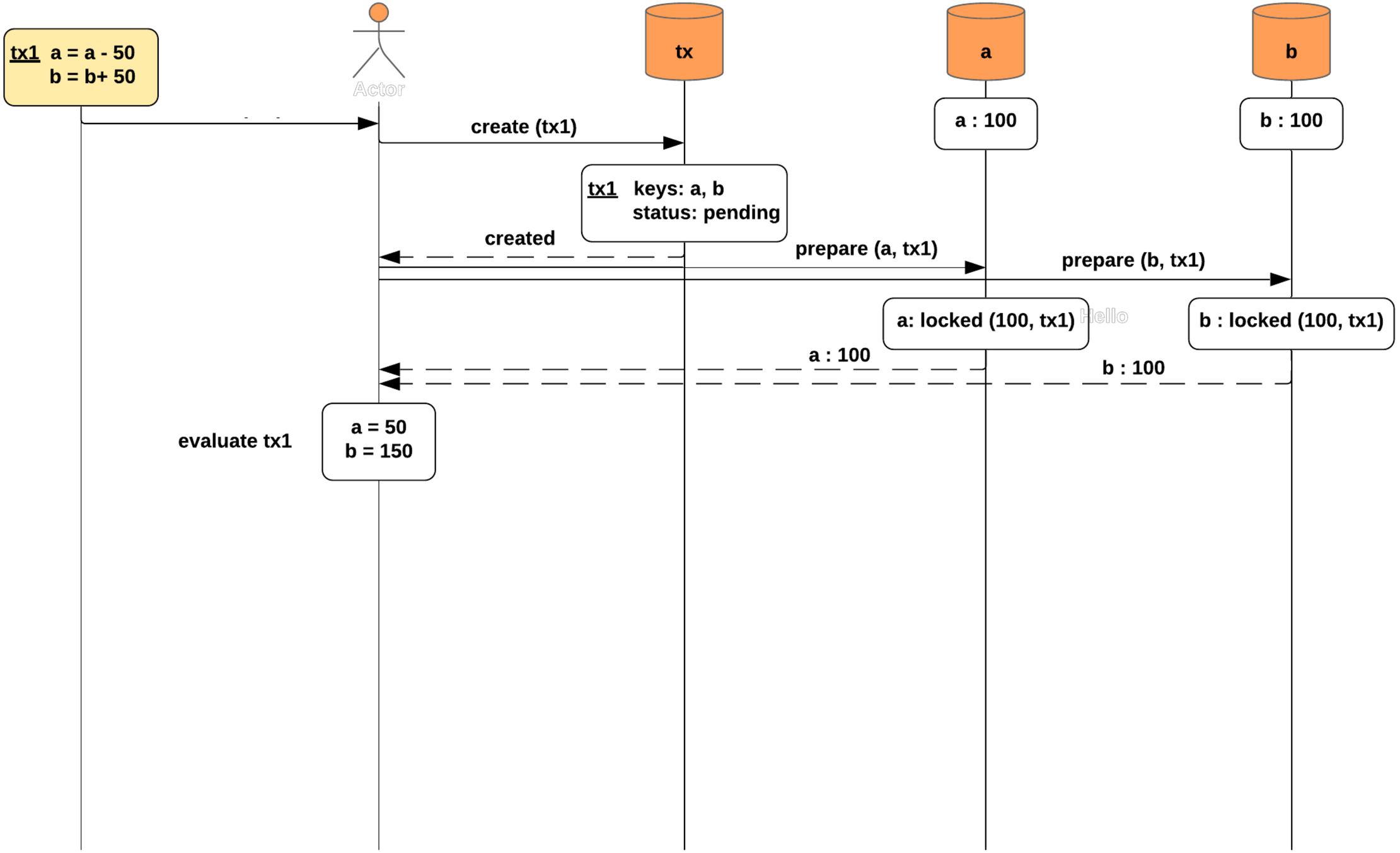
MSR-TR-2003-96

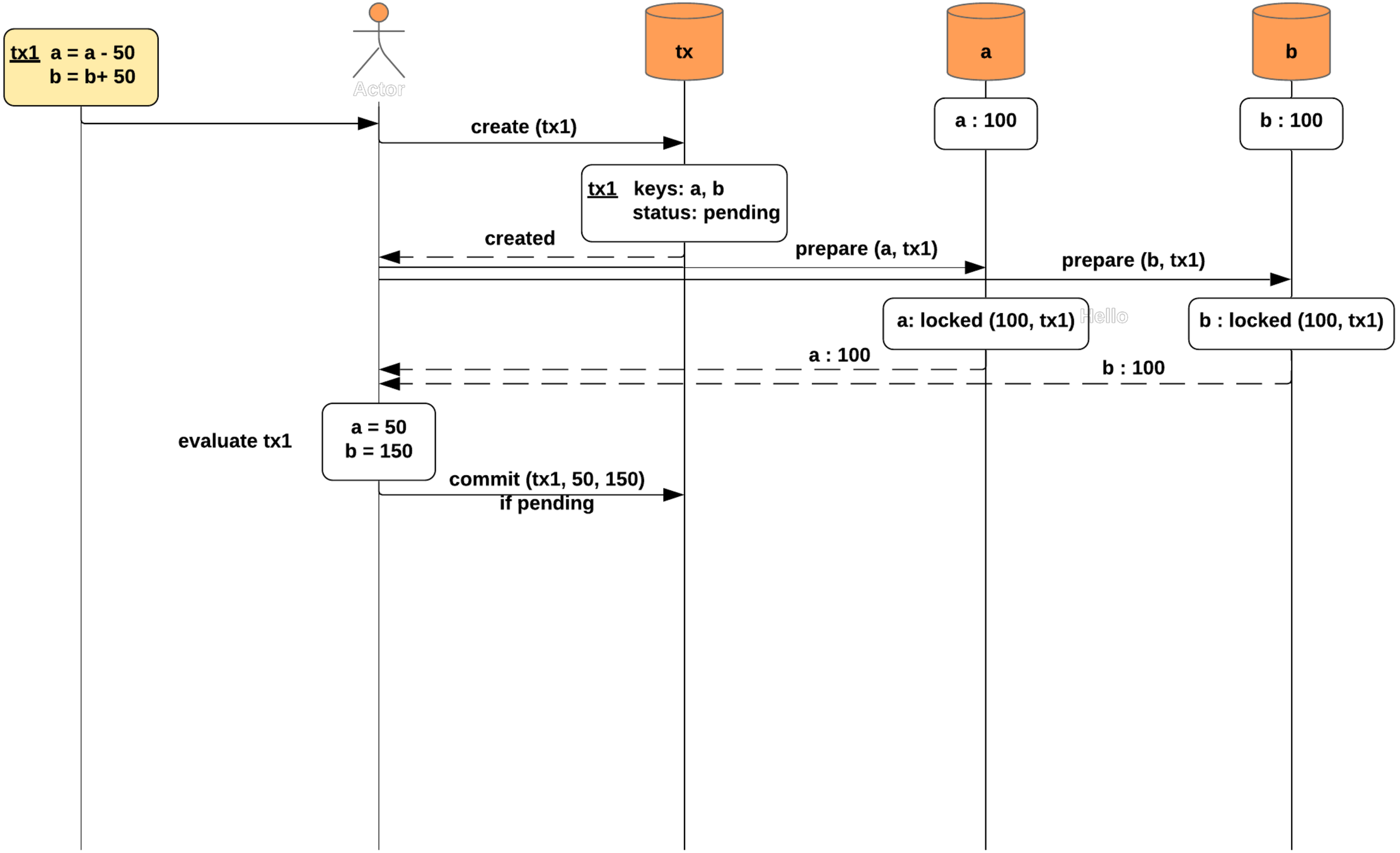
This paper appeared in *ACM Transactions on Database Systems*, Volume 31, Issue 1, March 2006 (pages 133-160). This version should differ from the published one only in formatting, except that it corrects one minor error on the last page.

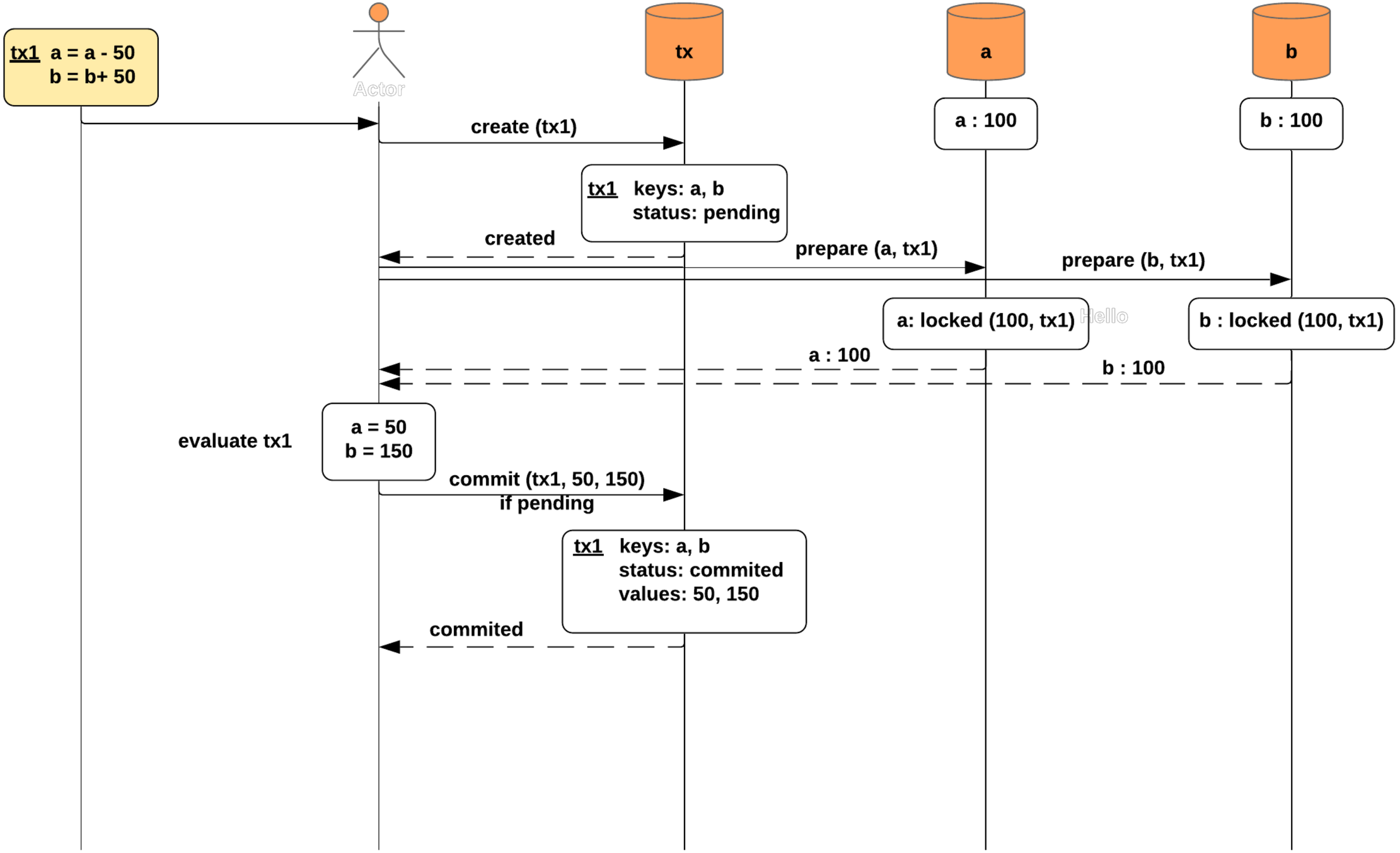

tx1 a = a - 50
b = b+ 50
Actor
tx
a
a : 100
b
b : 100

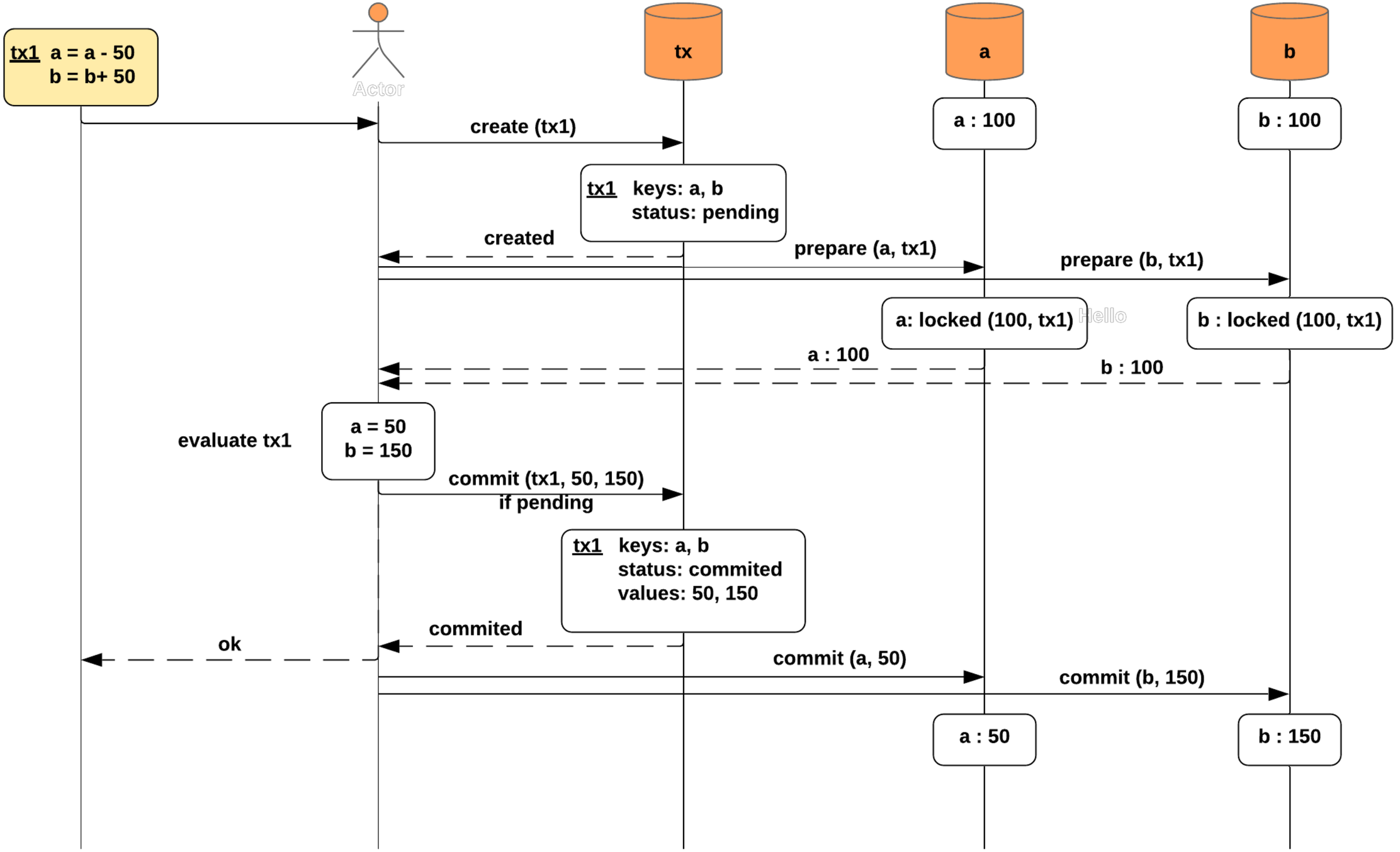

tx1 a = a - 50
b = b+ 50
Actor
tx
a
b
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created

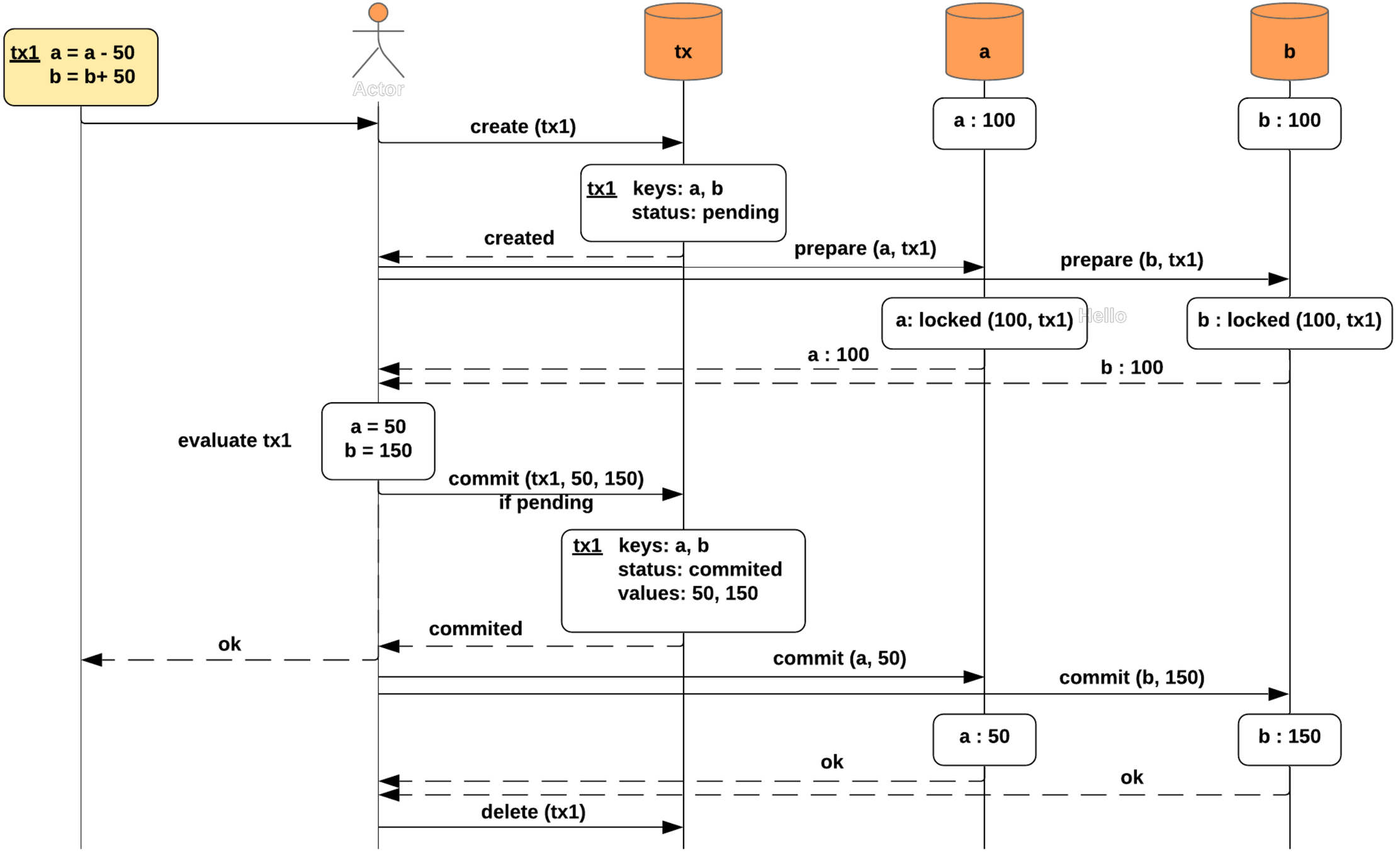

tx1 a = a - 50
b = b+ 50
Actor
tx
a
b
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created
prepare (a, tx1)
prepare (b, tx1)
Hello


tx1 a = a - 50
b = b+ 50
Actor
tx
a
b
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created
prepare (a, tx1)
prepare (b, tx1)
a: locked (100, tx1)
b : locked (100, tx1)
a : 100
b : 100
evaluate tx1
a = 50
b = 150

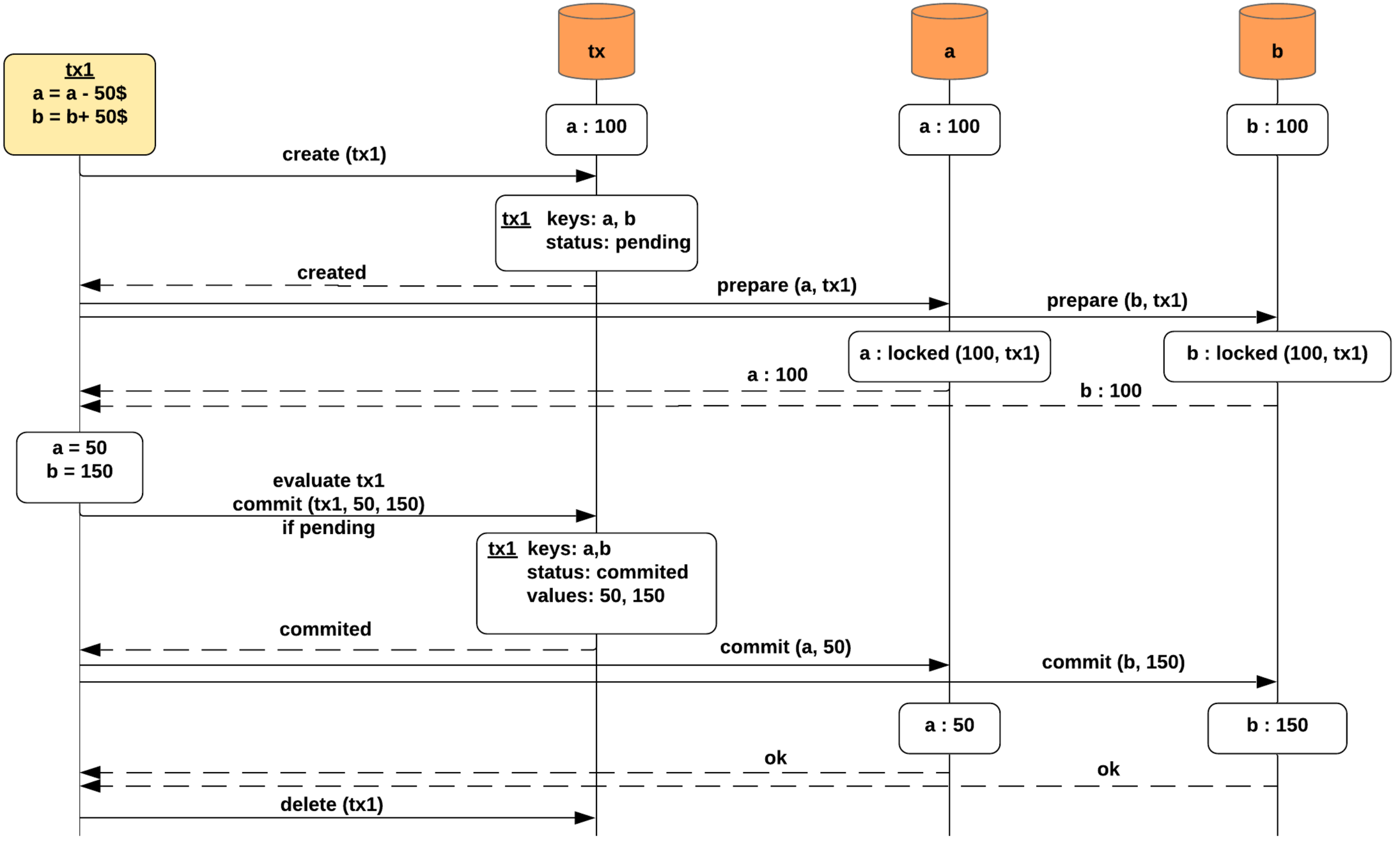

tx1 a = a - 50
b = b+ 50
Actor
tx
a
b
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created
prepare (a, tx1)
prepare (b, tx1)
a: locked (100, tx1)
b : locked (100, tx1)
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (tx1, 50, 150)
if pending


tx1 a = a - 50
b = b+ 50
Actor
tx
a
b
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created
prepare (a, tx1)
prepare (b, tx1)
a: locked (100, tx1)
Hello
b : locked (100, tx1)
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (tx1, 50, 150)
if pending
tx1 keys: a, b
status: commited
values: 50, 150
commited

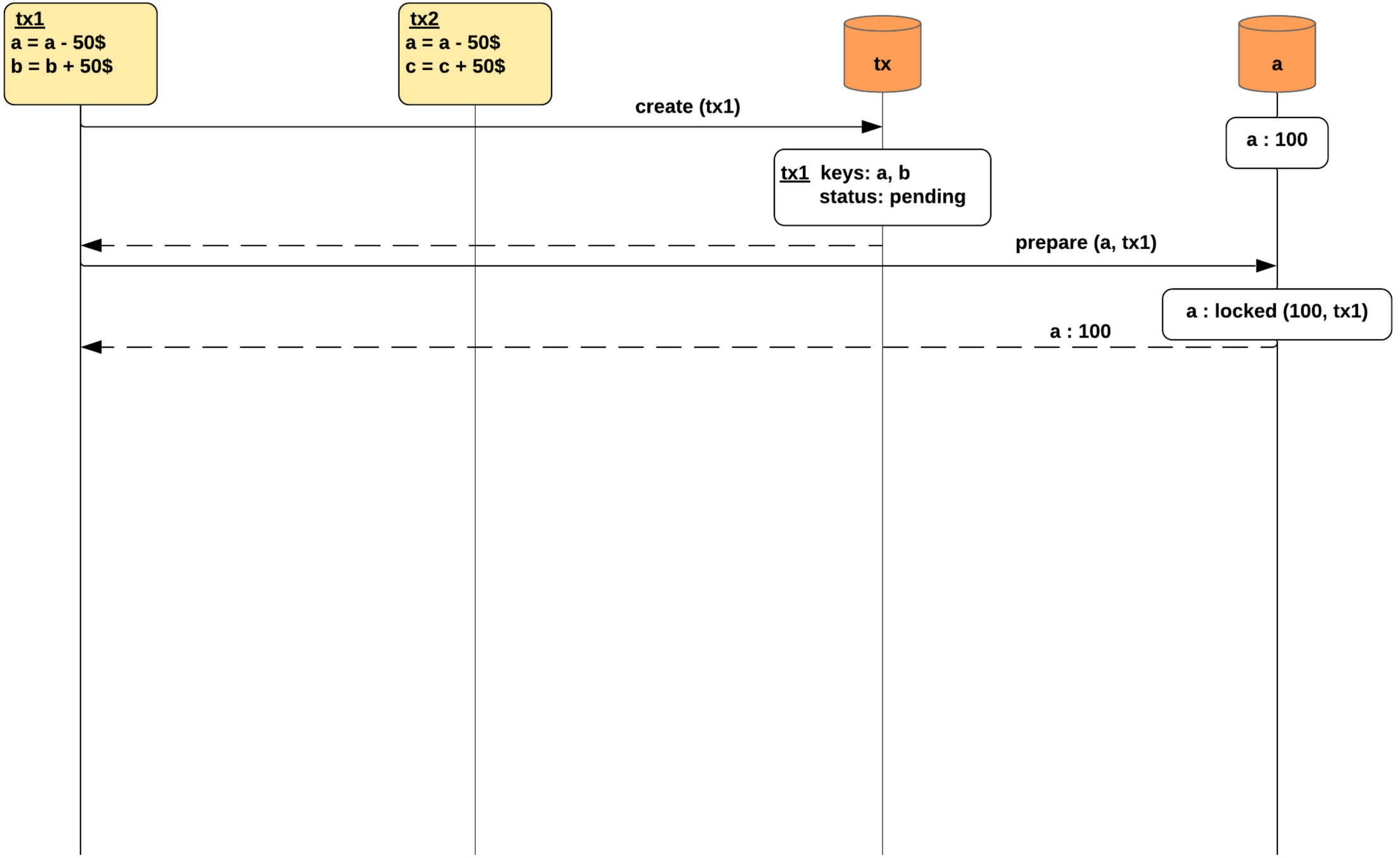

tx1 a = a - 50
b = b+ 50
Actor
tx
a
b
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created
prepare (a, tx1)
prepare (b, tx1)
a: locked (100, tx1)
Hello
b : locked (100, tx1)
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (tx1, 50, 150)
if pending
tx1 keys: a, b
status: commited
values: 50, 150
commited
ok
commit (a, 50)
commit (b, 150)
a : 50
b : 150

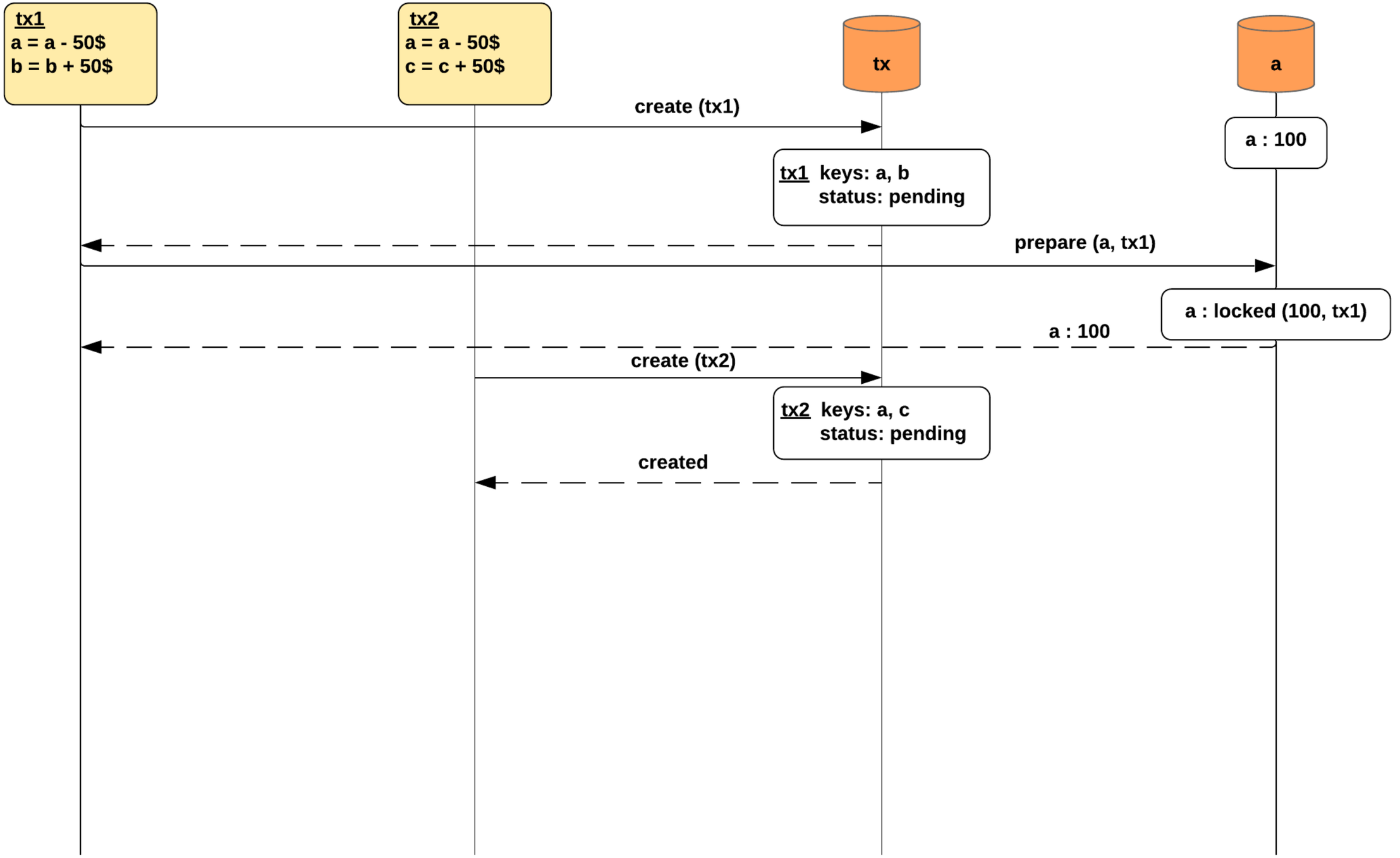

tx1 a = a - 50
b = b+ 50
Actor
tx
a
b
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created
prepare (a, tx1)
prepare (b, tx1)
a: locked (100, tx1)
Hello
b : locked (100, tx1)
a : 100
b : 100
evaluate tx1
a = 50
b = 150
commit (tx1, 50, 150)
if pending
tx1 keys: a, b
status: commited
values: 50, 150
commited
ok
commit (a, 50)
commit (b, 150)
a : 50
b : 150
ok
ok
delete (tx1)

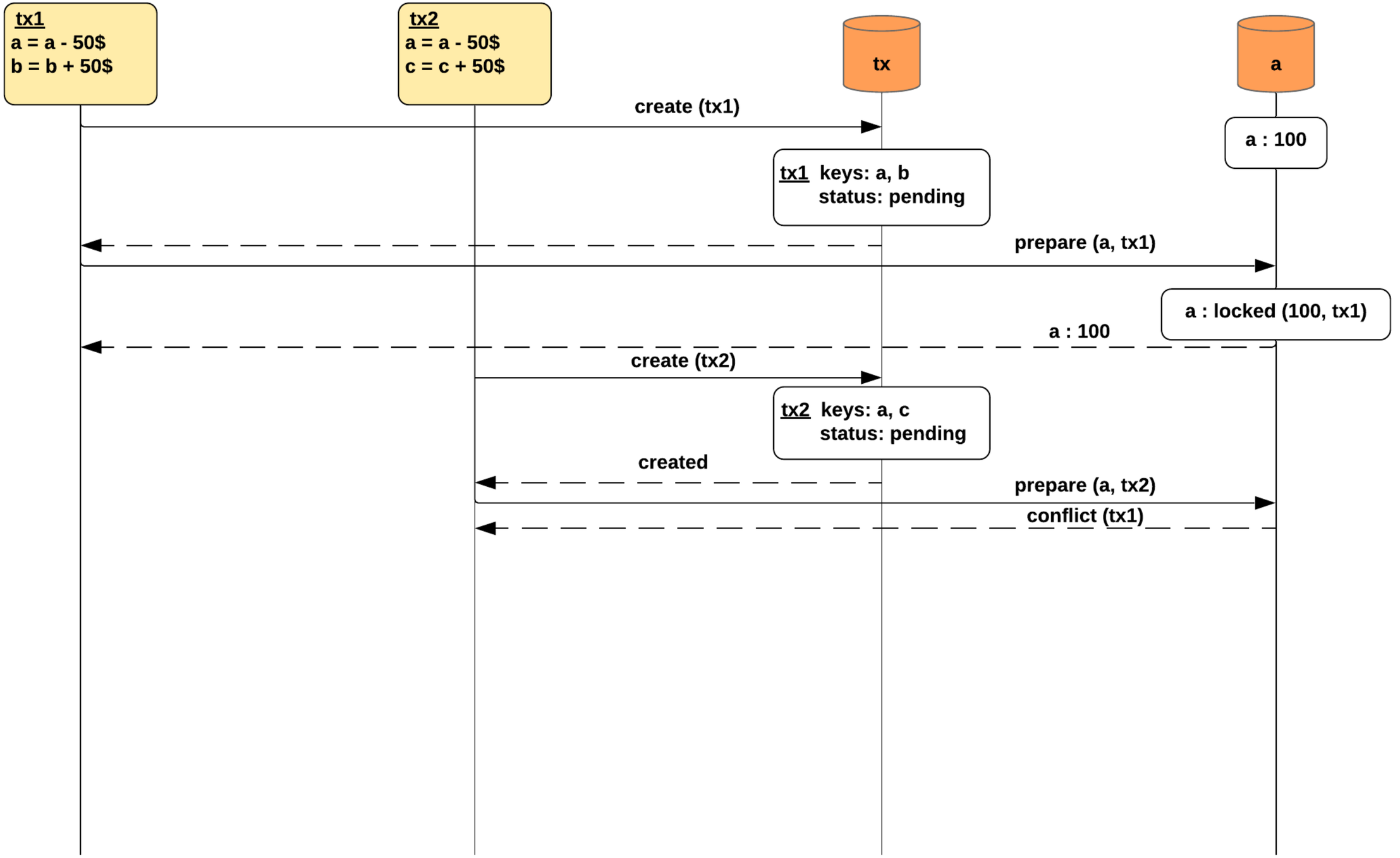
2PC	55%
Paxos Commit (PC)	88%

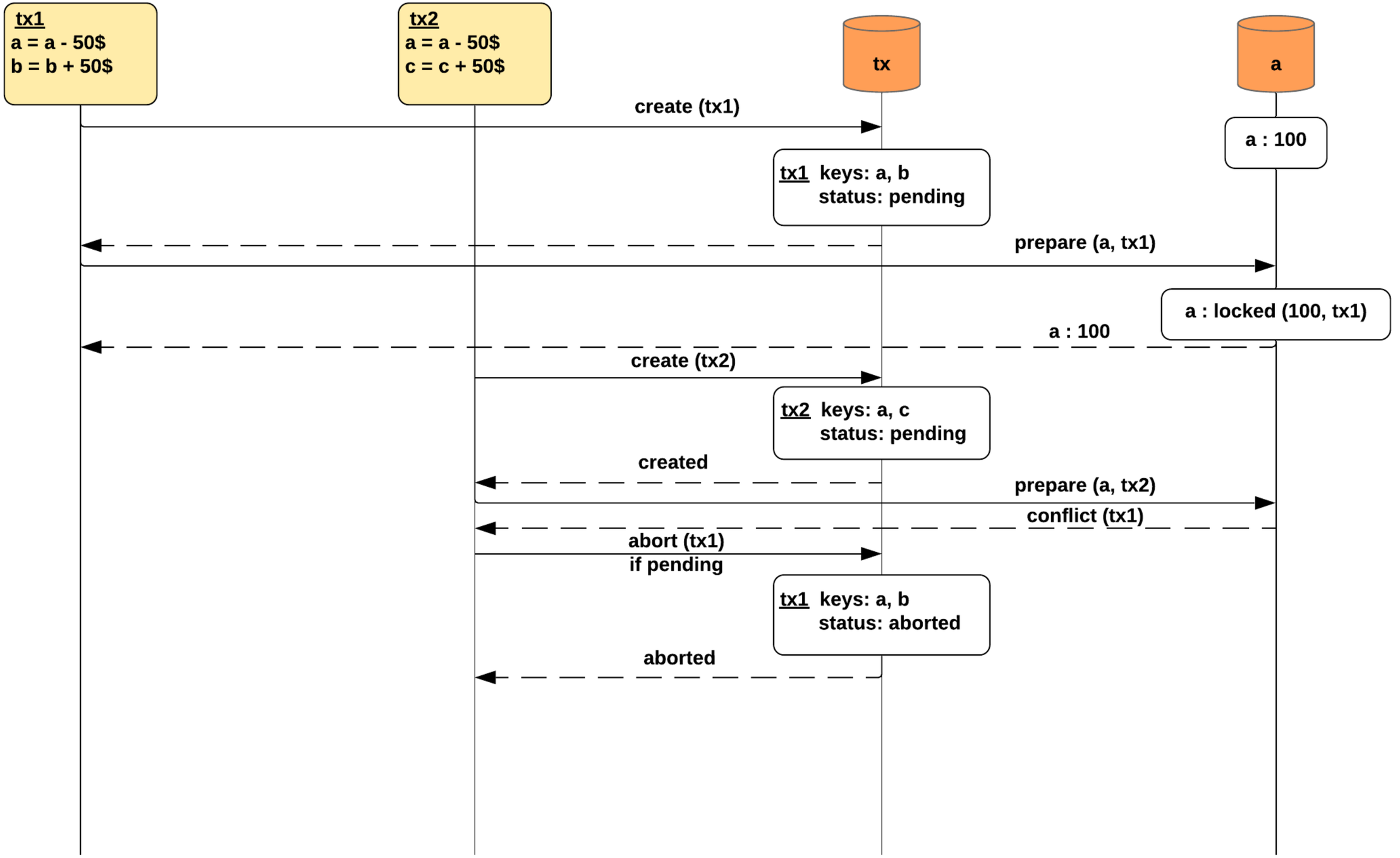

tx1
a = a - 50$
b = b+ 50$
tx
a
b
a : 100
a : 100
b : 100
create (tx1)
tx1 keys: a, b
status: pending
created
prepare (a, tx1)
prepare (b, tx1)
a : locked (100, tx1)
b : locked (100, tx1)
a : 100
b : 100
a = 50
b = 150
evaluate tx1
commit (tx1, 50, 150)
if pending
tx1 keys: a,b
status: commited
values: 50, 150
commited
commit (a, 50)
commit (b, 150)
a : 50
b : 150
ok
ok
delete (tx1)

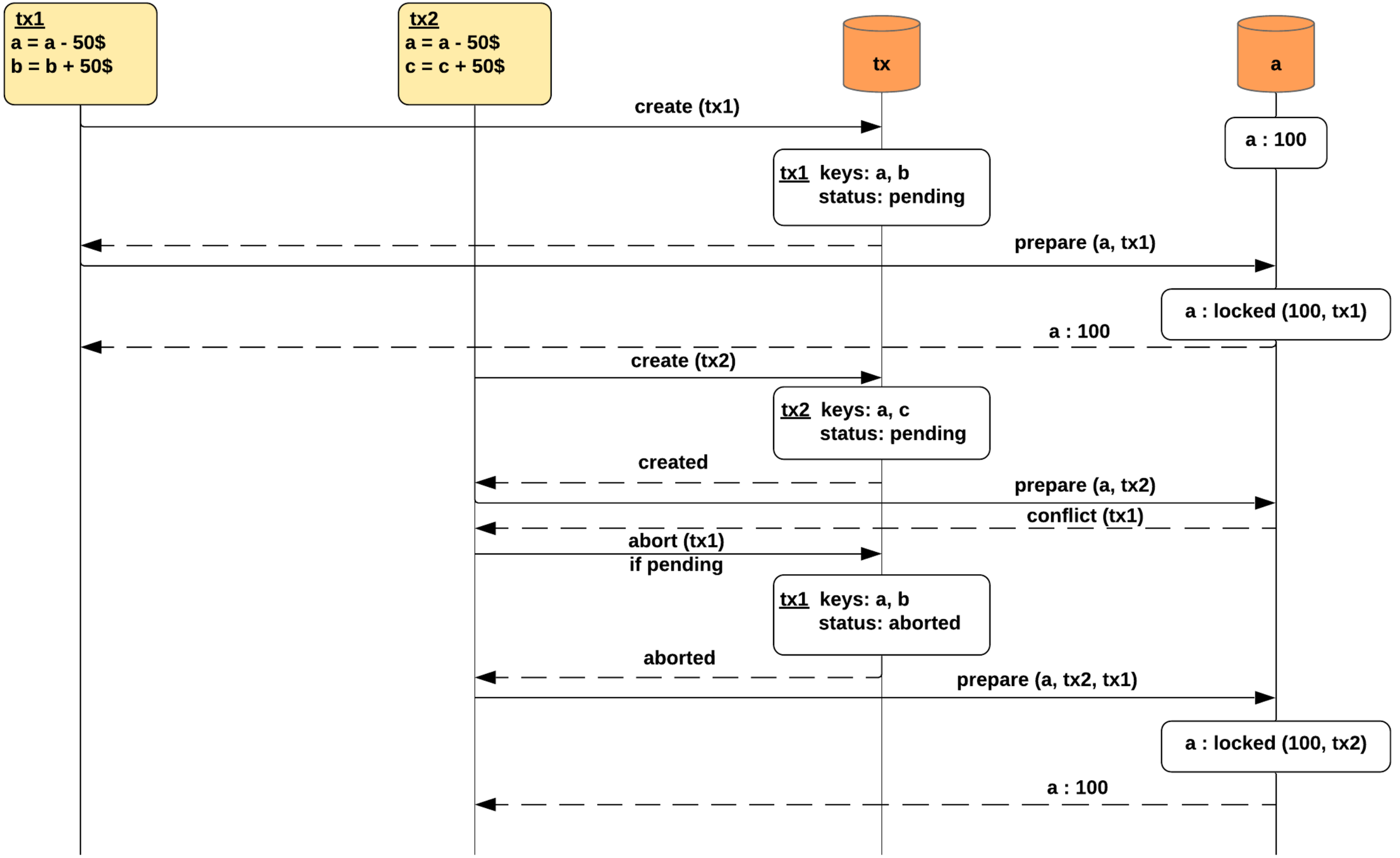
Conflicts

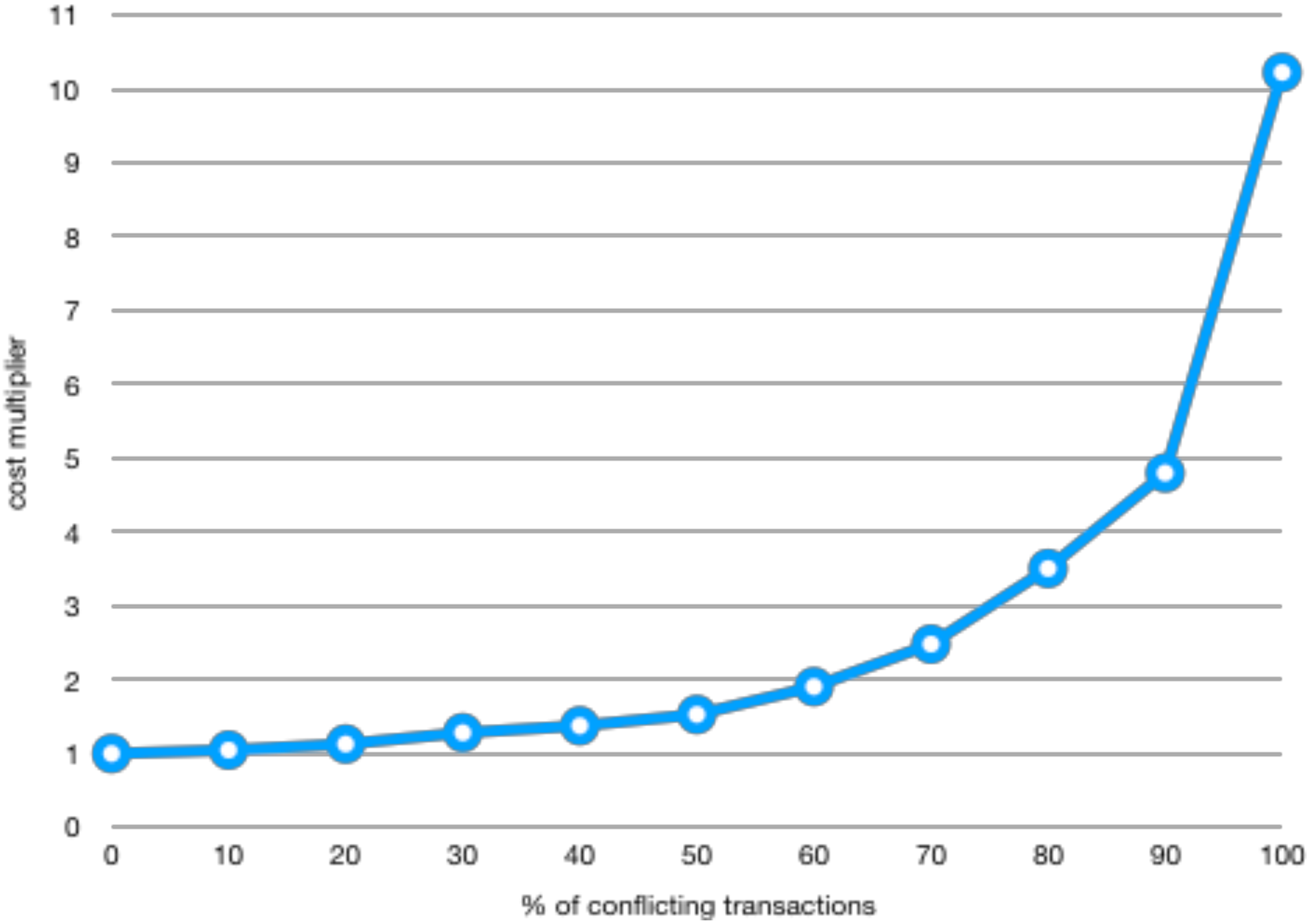

tx1
a = a - 50$
b = b + 50$
tx2
a = a - 50$
c = c + 50$
tx
a
create (tx1)
a : 100
tx1 keys: a, b
status: pending
prepare (a, tx1)
a : locked (100, tx1)
a : 100


tx1
a = a - 50$
b = b + 50$
tx2
a = a - 50$
c = c + 50$
tx
a
create (tx1)
a : 100
tx1 keys: a, b
status: pending
prepare (a, tx1)
a : locked (100, tx1)
a : 100
create (tx2)
tx2 keys: a, c
status: pending
created


tx1
a = a - 50$
b = b + 50$
tx2
a = a - 50$
c = c + 50$
tx
a
create (tx1)
a : 100
tx1 keys: a, b
status: pending
prepare (a, tx1)
a : locked (100, tx1)
a : 100
create (tx2)
tx2 keys: a, c
status: pending
created
prepare (a, tx2)
conflict (tx1)

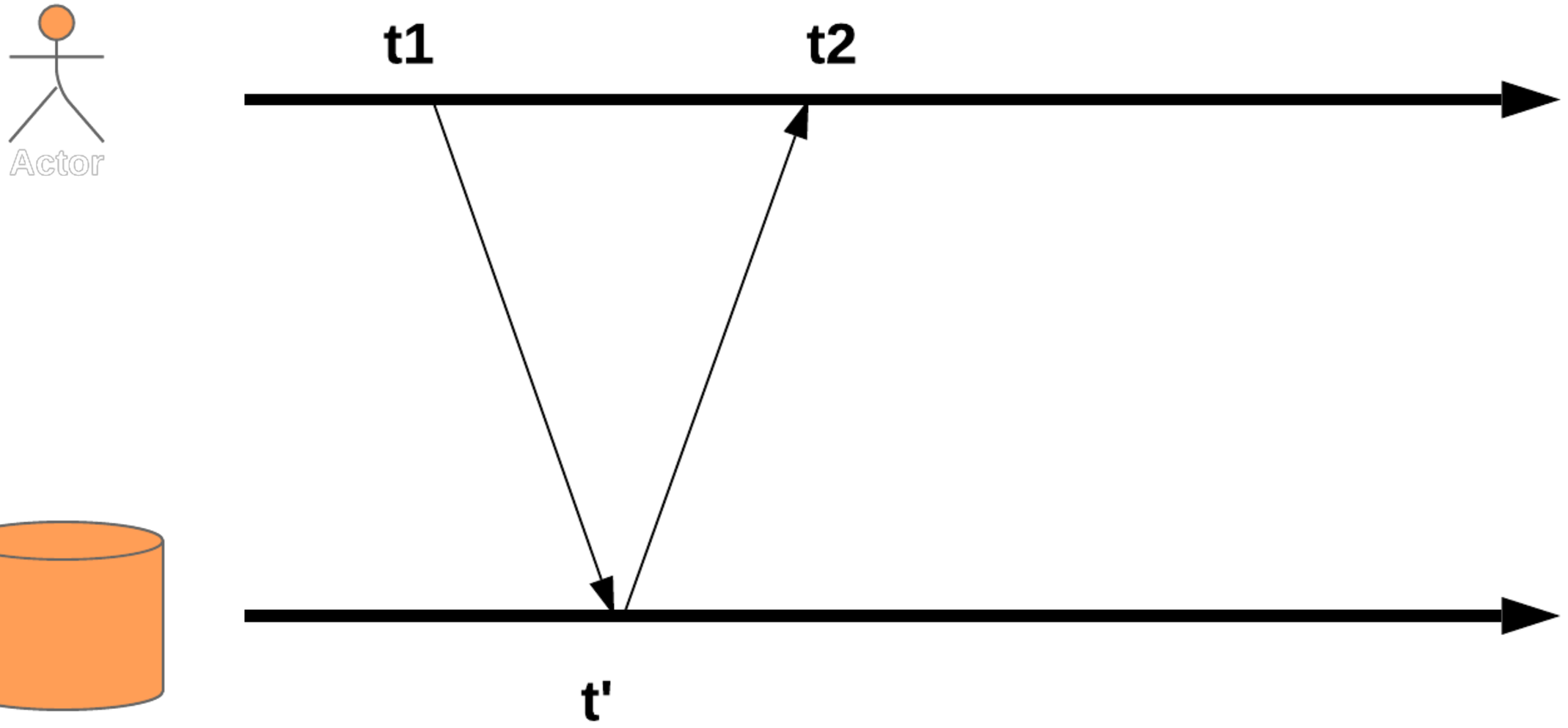

tx1
a = a - 50$
b = b + 50$
tx2
a = a - 50$
c = c + 50$
tx
a
create (tx1)
a : 100
tx1 keys: a, b
status: pending
prepare (a, tx1)
a : locked (100, tx1)
a : 100
create (tx2)
tx2 keys: a, c
status: pending
created
prepare (a, tx2)
conflict (tx1)
abort (tx1)
if pending
tx1 keys: a, b
status: aborted
aborted

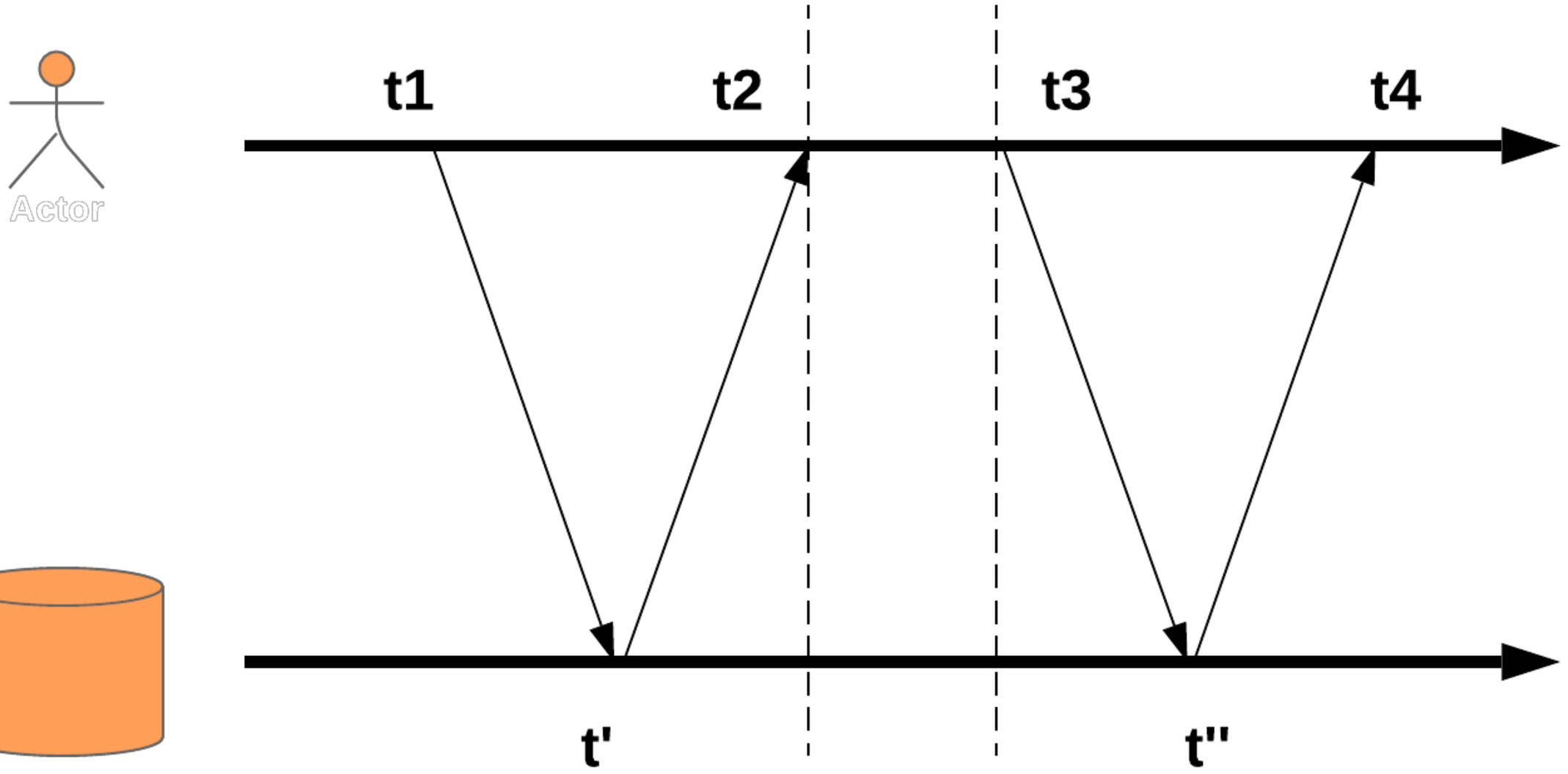

tx1
a = a - 50$
b = b + 50$
tx2
a = a - 50$
c = c + 50$
tx
a
create (tx1)
a : 100
tx1 keys: a, b
status: pending
prepare (a, tx1)
a : locked (100, tx1)
a : 100
create (tx2)
tx2 keys: a, c
status: pending
created
prepare (a, tx2)
conflict (tx1)
abort (tx1)
if pending
tx1 keys: a, b
status: aborted
aborted
prepare (a, tx2, tx1)
a : locked (100, tx2)
a : 100

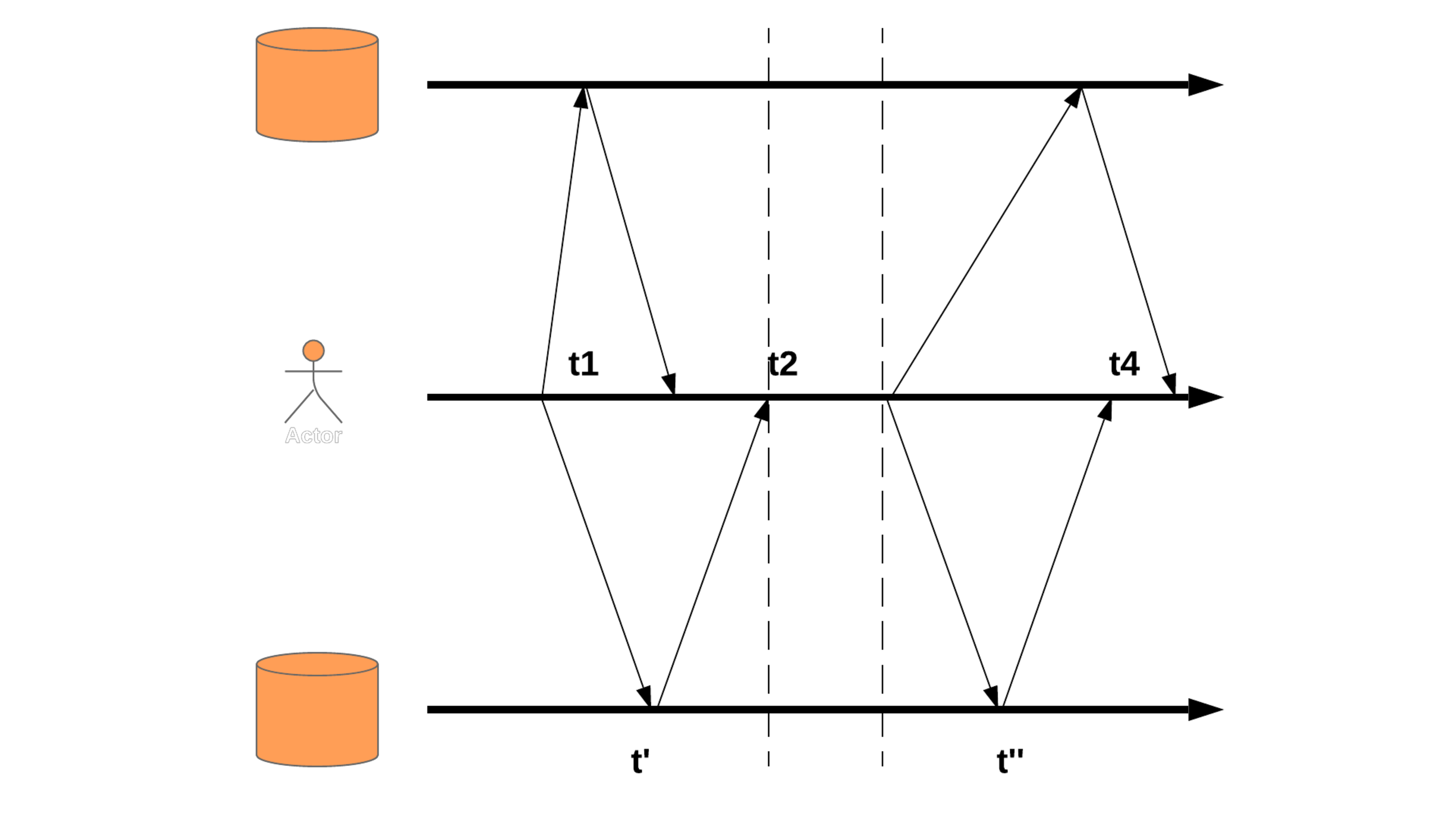

11
10
9
8
7
6
5
4
3
2
1
0
cost multiplier
0
10
20
30
40
50
60
70
80
90
100
% of conflicting transactions

Two Phase Commit	**availability and scalability**
Paxos Commit	**scalability** or **high cost on conflicts**

	with coordinator	clients as coordinators without contention	clients as coordinators with 100% contention
2PC	55%		
PC	88%	788%	4322%


Actor
t1
t2
t'


Actor
t1
t2
t3
t4
t'
t''


Actor
t1
t2
t4
t'
t''

	with coordinator	clients as coordinators without conflicts	clients as coordinators with 100% conflicts
2PC	55%		
PC	88%	788%	4322%
PC + Double reads		122%	555%

Eiger

To appear in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, April 2013

Stronger Semantics for Low-Latency Geo-Replicated Storage

Wyatt Lloyd*, Michael J. Freedman*, Michael Kaminsky†, and David G. Andersen‡
*Princeton University, †Intel Labs, ‡Carnegie Mellon University

Abstract

We present the first scalable, geo-replicated storage system that guarantees low latency, offers a rich data model, and provides "stronger" semantics. Namely, all client requests are satisfied in the local datacenter in which they arise; the system efficiently supports useful data model abstractions such as column families and counter columns; and clients can access data in a causally-consistent fashion with read-only and write-only transactional support, even for keys spread across many servers.

The primary contributions of this work are enabling scalable causal consistency for the complex column-family data model, as well as novel, non-blocking algorithms for both read-only and write-only transactions. Our evaluation shows that our system, Eiger, achieves low latency (single-ms), has throughput competitive with eventually-consistent and non-transactional Cassandra (less than 7% overhead for one of Facebook's real-world workloads), and scales out to large clusters almost linearly (averaging 96% increases up to 128 server clusters).

1 Introduction

Large-scale data stores are a critical infrastructure component of many Internet services. In this paper, we address the problem of building a geo-replicated data store targeted at applications that demand fast response times. Such applications are now common: Amazon, EBay, and Google all claim that a slight increase in user-perceived latency translates into concrete revenue loss [25, 26, 41, 50].

Providing *low latency* to the end-user requires two properties from the underlying storage system. First, storage nodes must be near the user to avoid long-distance round trip times; thus, data must be replicated geographically to handle users from diverse locations. Second, the storage layer itself must be fast: client reads and writes must be local to that nearby datacenter and not traverse the wide area. Geo-replicated storage also provides the important benefits of availability and fault tolerance.

Beyond low latency, many services benefit from a *rich data model*. Key-value storage—perhaps the simplest data model provided by data stores—is used by a number of services today [4, 29]. The simplicity of this data model, however, makes building a number of interesting services overly arduous, particularly compared to the column-family data models offered by systems like BigTable [19] and Cassandra [37]. These rich data models provide hierarchical sorted column-families and numerical counters. Column-families are well-matched to services such as Facebook, while counter columns are particularly useful for numerical statistics, as used by collaborative filtering (Digg, Reddit), likes (Facebook), or re-tweets (Twitter).

Unfortunately, to our knowledge, no existing geo-replicated data store provides guaranteed low latency, a rich column-family data model, and *stronger consistency semantics*: consistency guarantees stronger than the weakest choice—eventual consistency—and support for atomic updates and transactions. This paper presents Eiger, a system that achieves all three properties.

The consistency model Eiger provides is tempered by impossibility results: the strongest forms of consistency—such as linearizability, sequential, and serializability—are impossible to achieve with low latency [8, 42] (that is, latency less than the network delay between datacenters). Yet, some forms of stronger-than-eventual consistency are still possible and useful, e.g., *causal consistency* [2], and they can benefit system developers and users. In addition, *read-only* and *write-only transactions* that execute a batch of read or write operations at the same logical time can strengthen the semantics provided to a programmer.

Many previous systems satisfy two of our three design goals. Traditional databases, as well as the more recent Walter [52], MDCC [35], Megastore [9], and some Cassandra configurations, provide stronger semantics and a rich data model, but cannot guarantee low latency. Redis [48], CouchDB [23], and other Cassandra configurations provide low latency and a rich data model, but not stronger semantics. Our prior work on COPS [43] supports low latency, some stronger semantics—causal consistency and read-only transactions—but not a richer data model or write-only transactions (see §7.8 and §8 for a detailed comparison).

A key challenge of this work is to meet these three goals while *scaling* to a large numbers of nodes in a

$$\max_i \text{key}_i.\text{changeTime} \leq \min_i \text{key}_i.\text{nodeTime}$$

	with coordinator	clients as coordinators without conflicts	clients as coordinators with 100% conflicts
2PC	55%		
PC	88%	788%	4322%
PC + Double reads		122%	555%
PC + Eiger		111%	555%

Granola

Granola: Low-Overhead Distributed Transaction Coordination

James Cowling
MIT CSAIL

Barbara Liskov
MIT CSAIL

Abstract

This paper presents Granola, a transaction coordination infrastructure for building reliable distributed storage applications. Granola provides a strong consistency model, while significantly reducing transaction coordination overhead. We introduce specific support for a new type of *independent* distributed transaction, which we can serialize with no locking overhead and no aborts due to write conflicts. Granola uses a novel timestamp-based coordination mechanism to order distributed transactions, offering lower latency and higher throughput than previous systems that offer strong consistency.

Our experiments show that Granola has low overhead, is scalable and has high throughput. We implemented the TPC-C benchmark on Granola, and achieved 3× the throughput of a platform using a locking approach.

1 Introduction

Online storage systems run at very large scale and typically partition their state among many nodes to provide fast access and sufficient storage space. These systems need to provide persistence, availability, and good performance.

It is also highly desirable to run operations as *atomic transactions*, since this greatly simplifies the reasoning that application developers must do. Transactions allow users to ignore concurrency, since all operations appear to run sequentially in some serial order. Most distributed storage systems do not provide serializable transactions, however, because of concerns about performance and partition tolerance. Instead, they provide weaker semantics, e.g., eventual consistency [14] or causality [26].

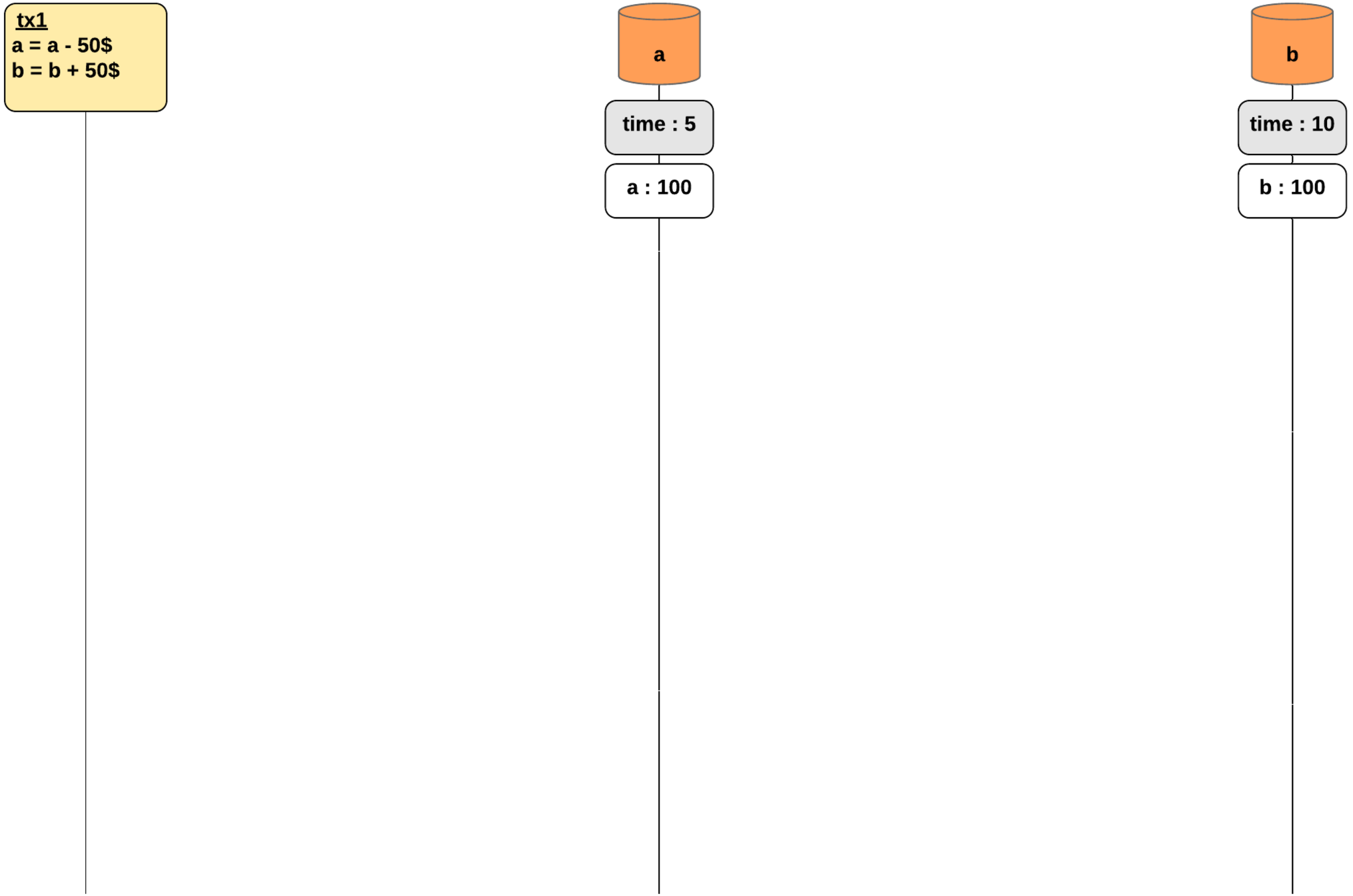
This paper presents Granola, an infrastructure for building distributed storage applications where data resides at multiple storage repositories. Granola supports atomic transactions, and provides serializability across all operations. Granola also provides persistence and high availability, along with low per-transaction overhead.

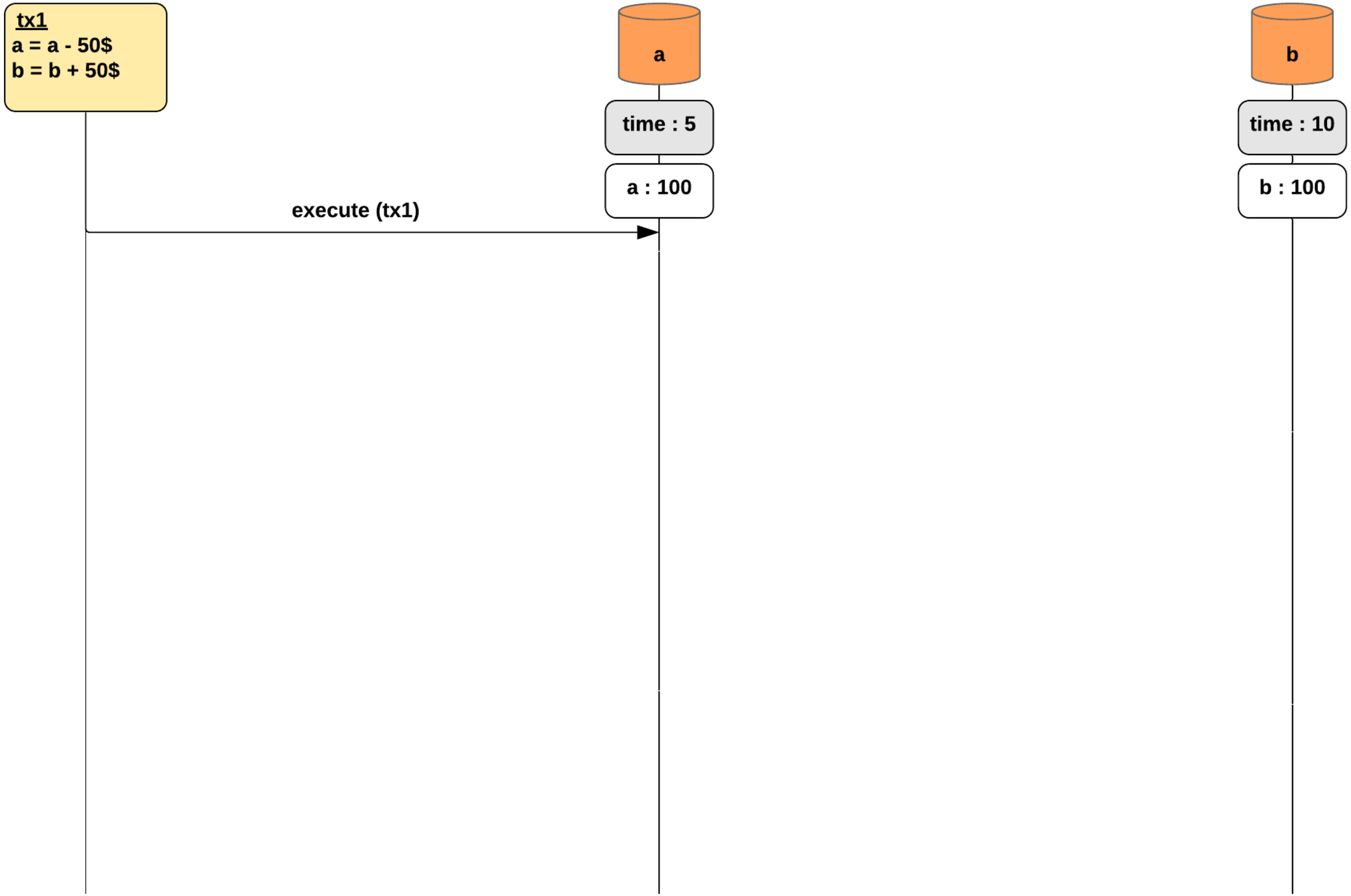
Granola provides transaction ordering, atomicity and reliability on behalf of storage applications that run on the platform. Applications specify their own operations, and Granola does not interpret operation semantics. Granola can thus be used to support a wide variety of storage systems, such as databases and object stores. Granola implements atomic *one-round* transactions. These execute in one round of communication between a user and the storage system, and are used extensively in online transaction processing workloads to avoid the cost of user stalls [7,20,32].

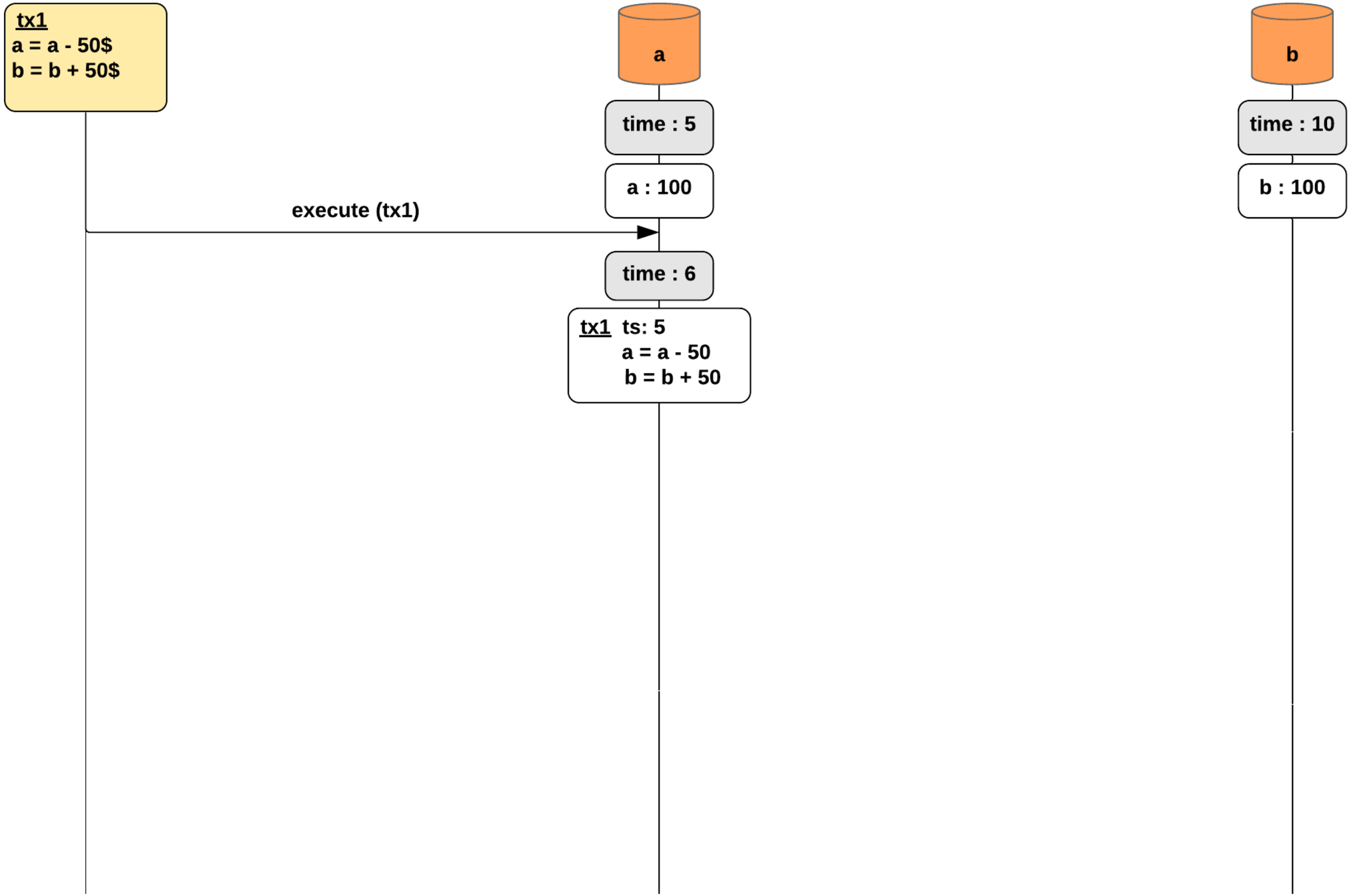
Granola supports three classes of one-round transactions. *Single-repository transactions* execute on a single storage node; we expect that most transactions will be in this class, since data is likely to be well-partitioned. *Coordinated distributed transactions* execute atomically across multiple storage nodes, and commit only if all participants vote to commit; these transactions are what is provided by traditional two-phase commit. We also support a new transaction class, which we term *independent distributed transactions*. These execute atomically across a set of nodes, but do not require agreement, since each participant will independently come to the same commit decision. Examples include an operation to give everyone a raise, an atomic update of a replicated table, or a read-only query that obtains a snapshot of distributed tables.

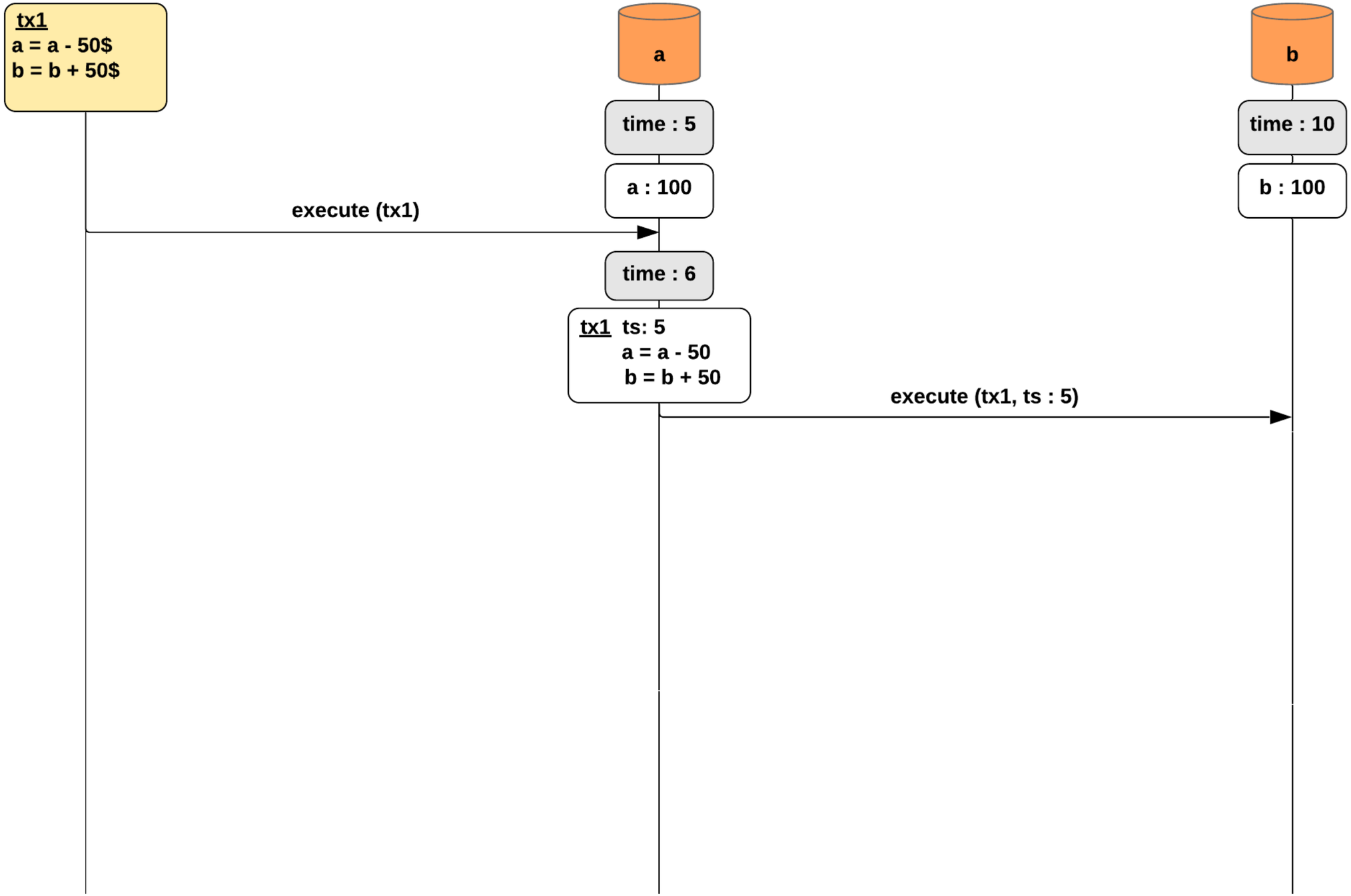
Granola uses a timestamp-based coordination mechanism to provide serializability for single-repository and independent transactions *without* locking, using clients to propagate timestamp ordering constraints between repositories. This provides a substantial reduction in overhead from locking, log management and aborts, and a consequent improvement in throughput. Granola provides this lock-free coordination protocol while handling single-repository and independent transactions, and adopts a lock-based protocol when handling coordinated transactions. Granola's throughput is similar to existing state-of-the-art approaches when operating under the locking protocol, but significantly higher when it is not.

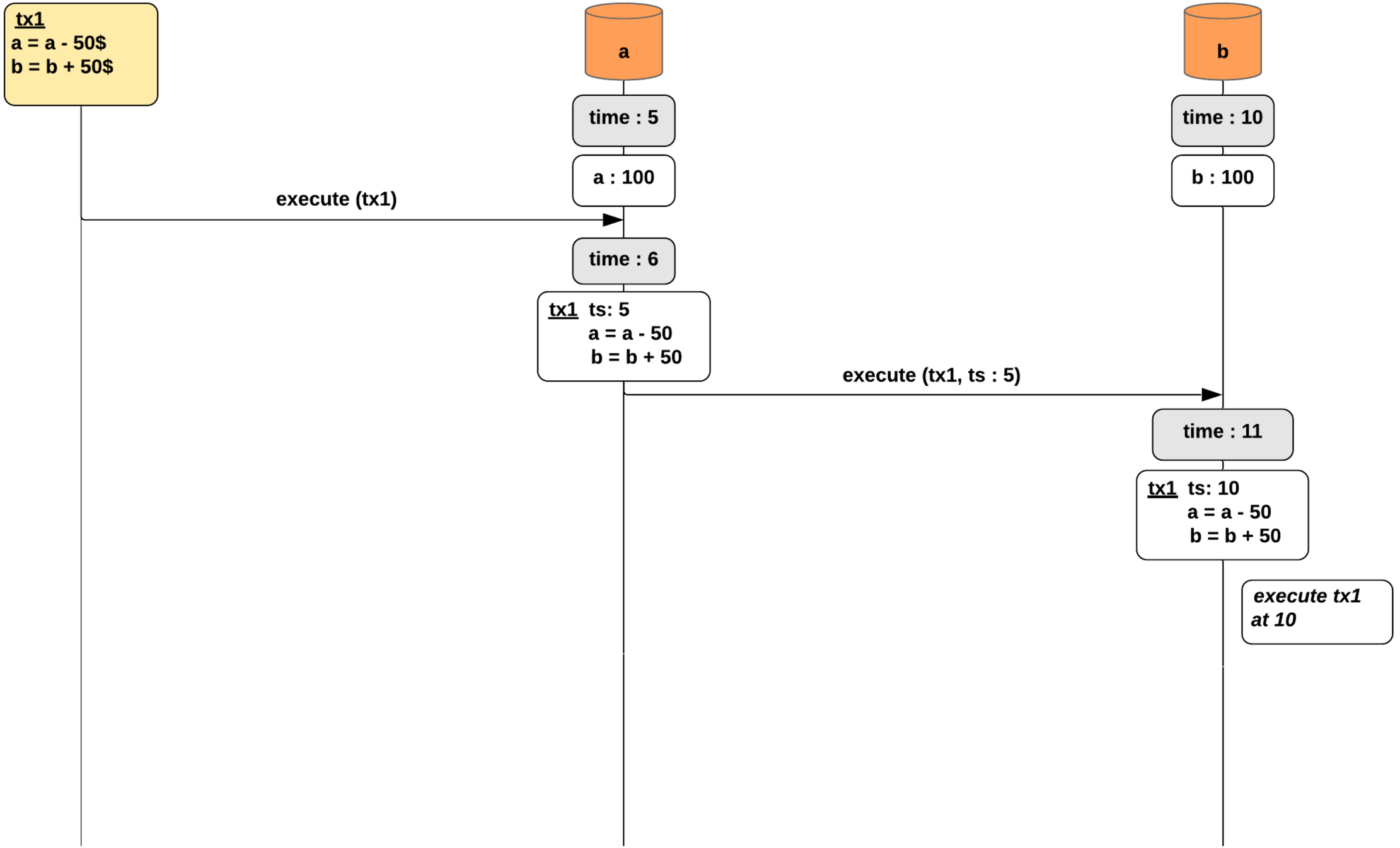
Granola provides low latency for all transaction types:

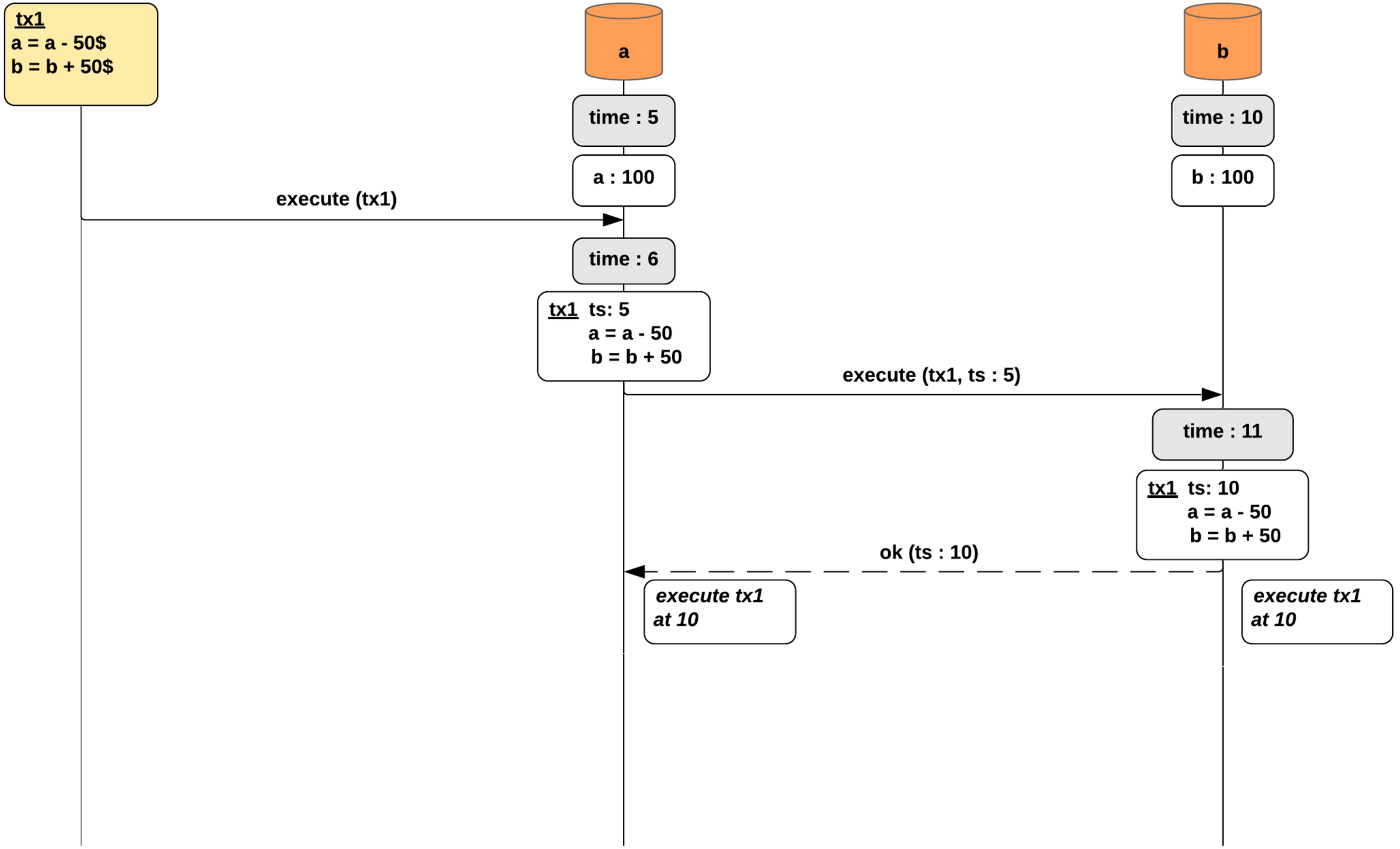

tx1
a = a - 50$
b = b + 50$
a
time : 5
a : 100
b
time : 10
b : 100

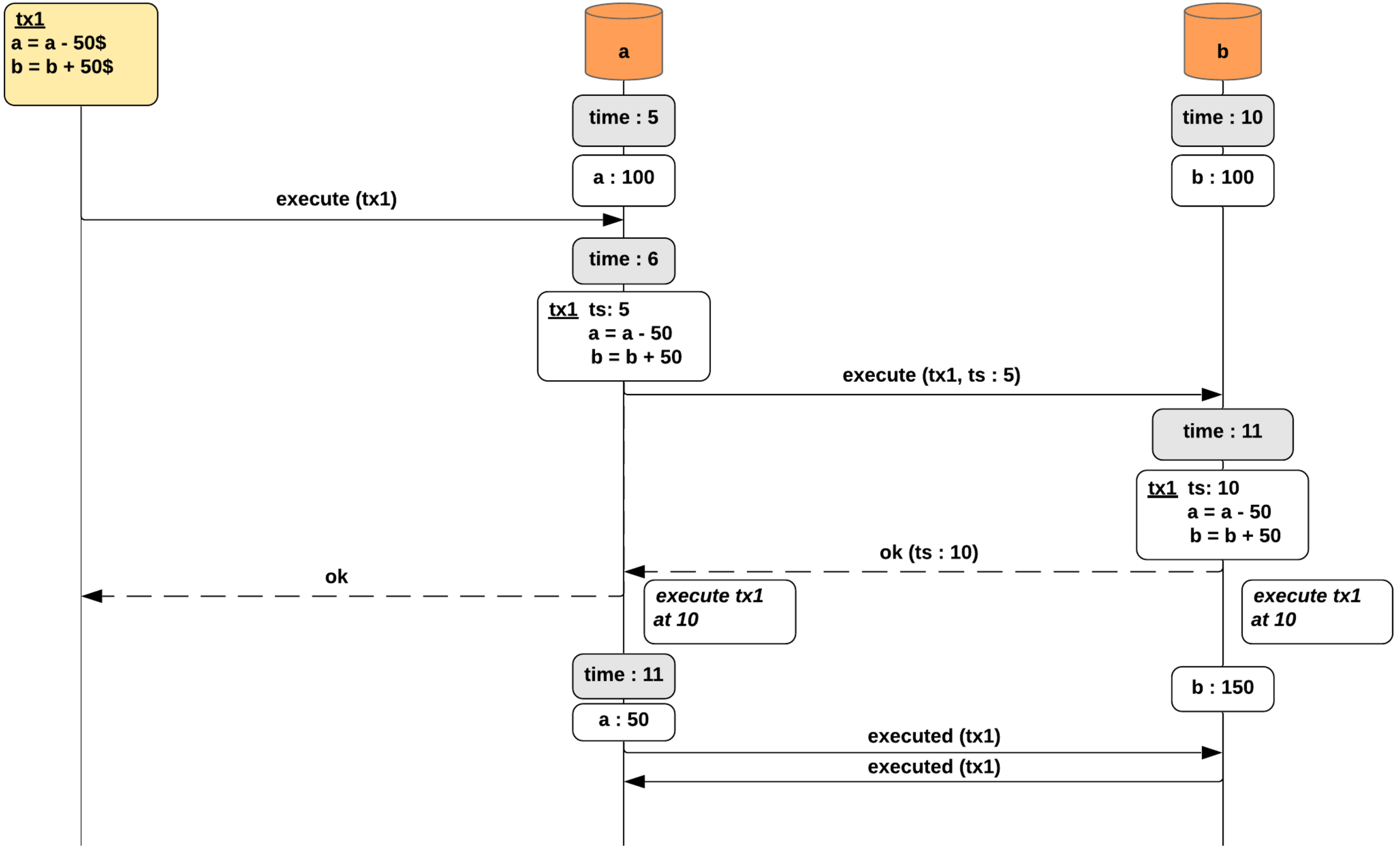

tx1
a = a - 50$
b = b + 50$
a
time : 5
a : 100
b
time : 10
b : 100
execute (tx1)

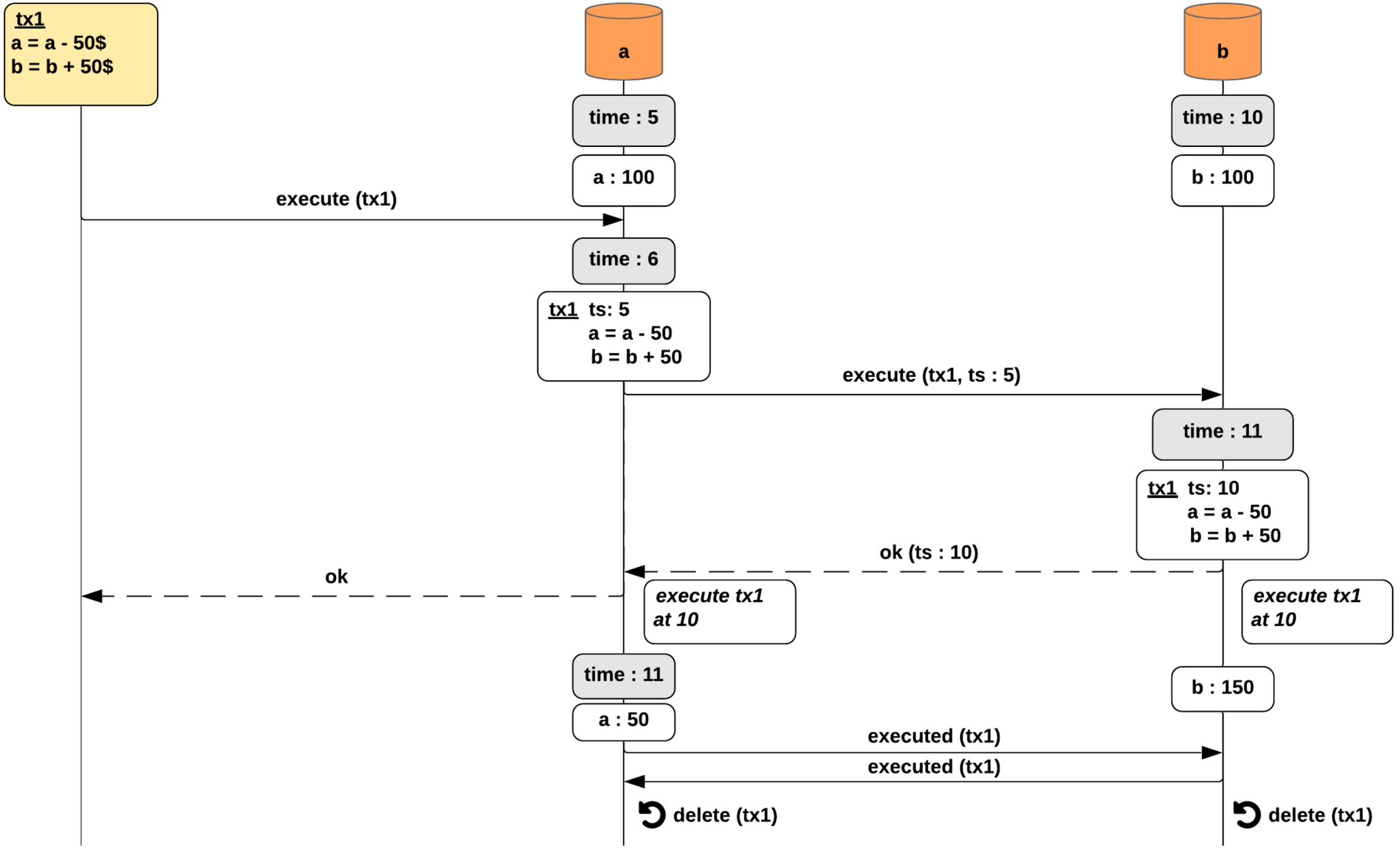

tx1
a = a - 50$
b = b + 50$
a
b
time : 5
time : 10
a : 100
b : 100
execute (tx1)
time : 6
tx1 ts: 5
a = a - 50
b = b + 50


tx1
a = a - 50$
b = b + 50$
a
b
time : 5
time : 10
a : 100
b : 100
execute (tx1)
time : 6
tx1 ts: 5
a = a - 50
b = b + 50
execute (tx1, ts : 5)

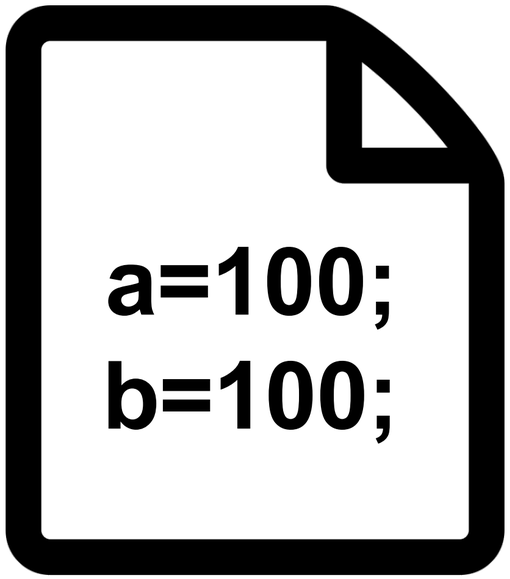

tx1
a = a - 50$
b = b + 50$
a
b
time : 5
time : 10
a : 100
b : 100
execute (tx1)
time : 6
tx1 ts: 5
a = a - 50
b = b + 50
execute (tx1, ts : 5)
time : 11
tx1 ts: 10
a = a - 50
b = b + 50
execute tx1
at 10

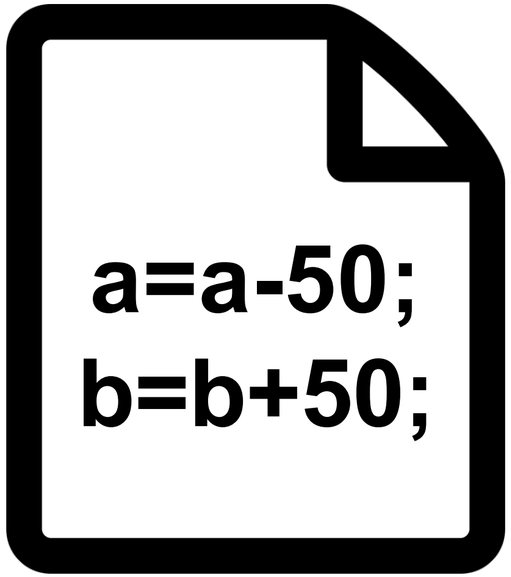

tx1
a = a - 50$
b = b + 50$
a
b
time : 5
time : 10
a : 100
b : 100
execute (tx1)
time : 6
tx1 ts: 5
a = a - 50
b = b + 50
execute (tx1, ts : 5)
time : 11
tx1 ts: 10
a = a - 50
b = b + 50
ok (ts : 10)
execute tx1
at 10
execute tx1
at 10

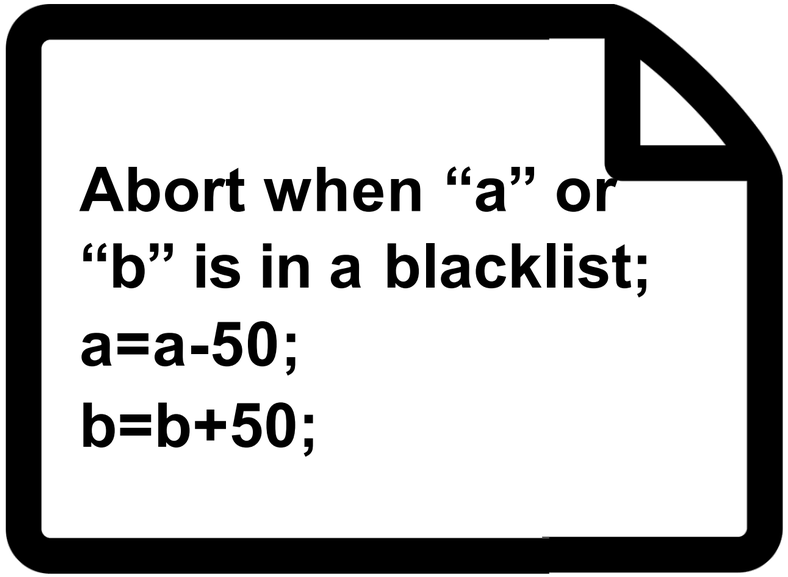

tx1
a = a - 50$
b = b + 50$
a
b
time : 5
time : 10
a : 100
b : 100
execute (tx1)
time : 6
tx1 ts: 5
a = a - 50
b = b + 50
execute (tx1, ts : 5)
time : 11
tx1 ts: 10
a = a - 50
b = b + 50
ok (ts : 10)
ok
execute tx1
at 10
execute tx1
at 10
time : 11
b : 150
a : 50
executed (tx1)
executed (tx1)


tx1
a = a - 50$
b = b + 50$
a
b
time : 5
time : 10
a : 100
b : 100
execute (tx1)
time : 6
tx1 ts: 5
a = a - 50
b = b + 50
execute (tx1, ts : 5)
time : 11
tx1 ts: 10
a = a - 50
b = b + 50
ok (ts : 10)
ok
execute tx1
at 10
execute tx1
at 10
time : 11
b : 150
a : 50
executed (tx1)
executed (tx1)
delete (tx1)
delete (tx1)

Independent transactions





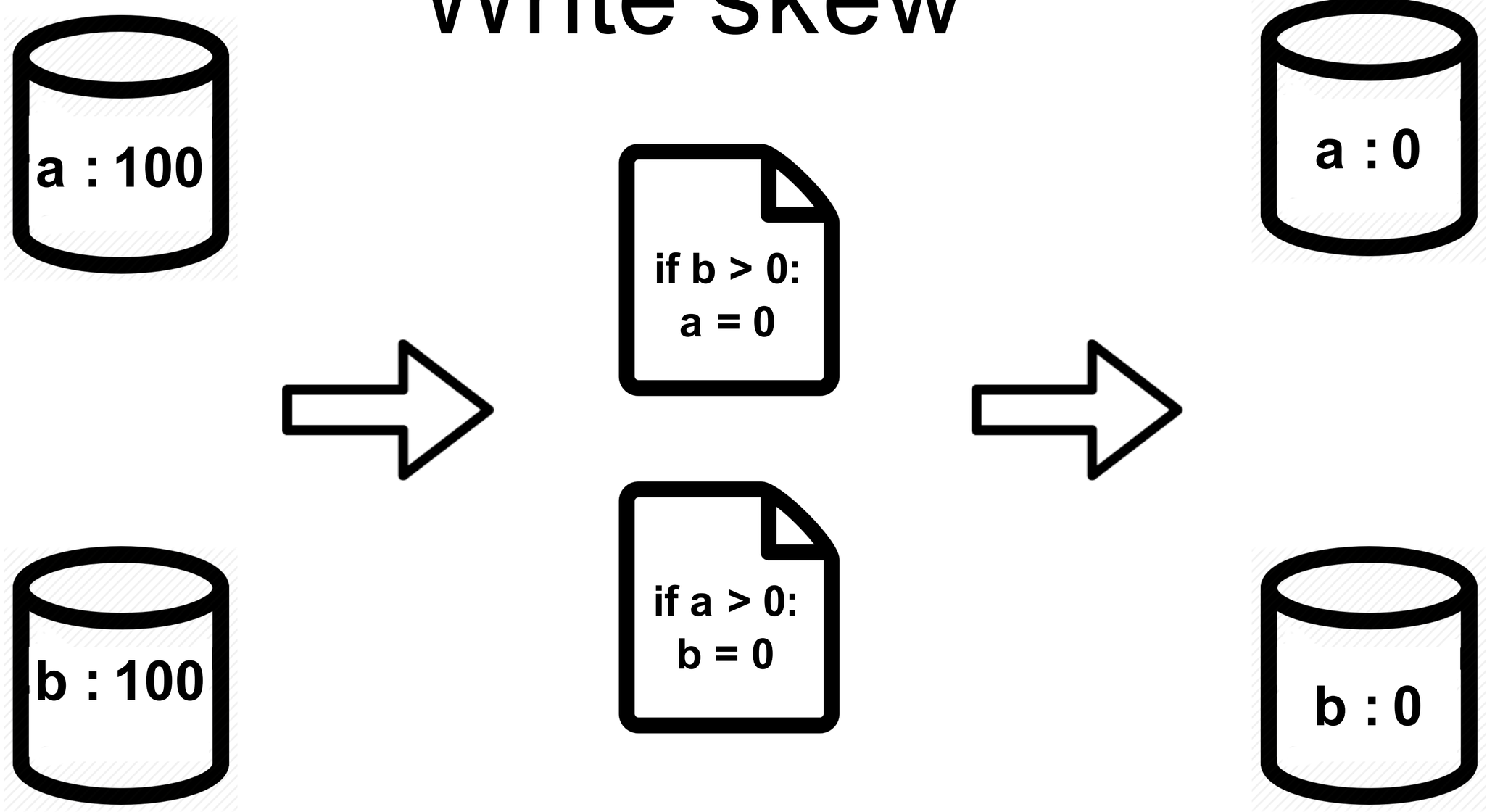


	with coordinator	clients as coordinators without conflicts	clients as coordinators with 100% conflicts
2PC	55%		
PC	88%	788%	4322%
PC + Double reads		122%	555%
PC + Eiger		111%	555%
Granola		77%	77%

Correctness vs Performance

Isolation	Stale read	Write skew	Lost update	Read skew
Strict serializability				
Serializability	✓			
Snapshot isolation	✓	✓		
Read Atomic	✓	✓	✓	
Read Committed	✓	✓	✓	✓

Write skew



Snapshot isolation

Snapshot isolation

Execute a transaction unless global condition changed:

- please do X if I'm still a leader

Snapshot isolation

Execute a transaction unless global condition changed:

- please do X if I'm still a leader
- please do X if a cached value hasn't changed

- Previous workload: transfers between two accounts and reading three accounts.

- Previous workload: transfers between two accounts and reading three accounts.
- Each initiator of a transaction caches a blacklist

- Previous workload: transfers between two accounts and reading three accounts.
- Each initiator of a transaction caches a blacklist
- If any of the accounts in a transaction is in the blacklist the transaction is rejected before its even started

- Previous workload: transfers between two accounts and reading three accounts.
- Each initiator of a transaction caches a blacklist
- If any of the accounts in a transaction is in the blacklist the transaction is rejected before its even started
- We want to avoid a divergence between the cached blacklist and the actual blacklist to prevent fraud ASAP

- Previous workload: transfers between two accounts and reading three accounts.
- Each initiator of a transaction caches a blacklist
- If any of the accounts in a transaction is in the blacklist the transaction is rejected before its even started
- We want to avoid a divergence between the cached blacklist and the actual blacklist to prevent fraud ASAP
- A solution is to include a condition on the blacklist's version into all transactions

	with coordinator	clients as coordinators without conflicts	clients as coordinators with 100% conflicts
2PC	81%		
PC	100%	5218%	4854%
PC + Double reads		1009%	1336%
PC + Eiger		990%	1045%
Granola		118%	118%

Percolator

Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek
dpeng@google.com, fdabek@google.com
Google, Inc.

Abstract

Updating an index of the web as documents are crawled requires continuously transforming a large repository of existing documents as new documents arrive. This task is one example of a class of data processing tasks that transform a large repository of data via small, independent mutations. These tasks lie in a gap between the capabilities of existing infrastructure. Databases do not meet the storage or throughput requirements of these tasks: Google's indexing system stores tens of petabytes of data and processes billions of updates per day on thousands of machines. MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency.

We have built Percolator, a system for incrementally processing updates to a large data set, and deployed it to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, we process the same number of documents per day, while reducing the average age of documents in Google search results by 50%.

1 Introduction

Consider the task of building an index of the web that can be used to answer search queries. The indexing system starts by crawling every page on the web and processing them while maintaining a set of invariants on the index. For example, if the same content is crawled under multiple URLs, only the URL with the highest PageRank [28] appears in the index. Each link is also inverted so that the anchor text from each outgoing link is attached to the page the link points to. Link inversion must work across duplicates: links to a duplicate of a page should be forwarded to the highest PageRank duplicate if necessary.

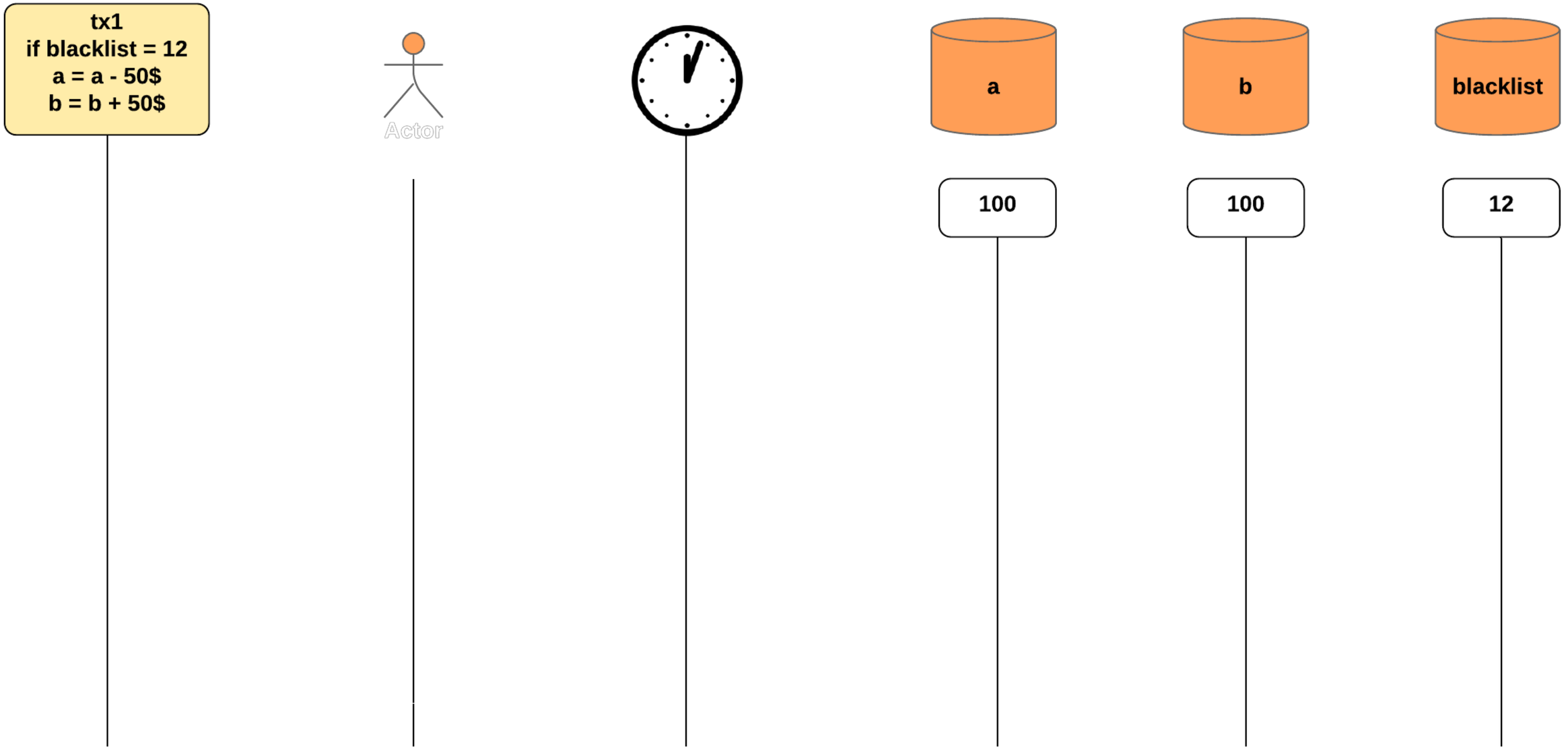
This is a bulk-processing task that can be expressed as a series of MapReduce [13] operations: one for clustering duplicates, one for link inversion, etc. It's easy to maintain invariants since MapReduce limits the parallelism of the computation; all documents finish one processing step before starting the next. For example, when the indexing system is writing inverted links to the current highest-PageRank URL, we need not worry about its PageRank concurrently changing; a previous MapReduce step has already determined its PageRank.

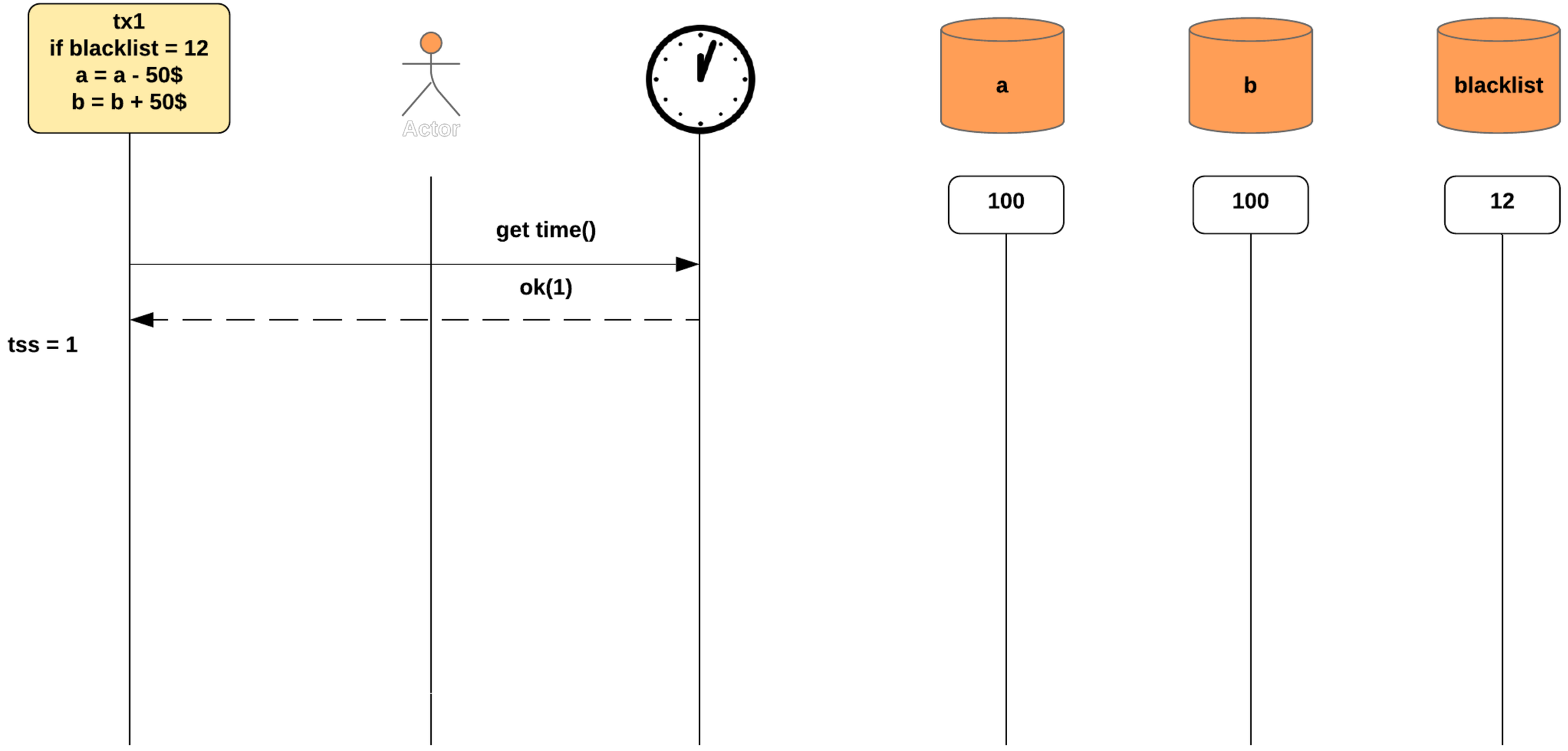
Now, consider how to update that index after recrawling some small portion of the web. It's not sufficient to run the MapReduces over just the new pages since, for example, there are links between the new pages and the rest of the web. The MapReduces must be run again over the entire repository, that is, over both the new pages and the old pages. Given enough computing resources, MapReduce's scalability makes this approach feasible, and, in fact, Google's web search index was produced in this way prior to the work described here. However, reprocessing the entire web discards the work done in earlier runs and makes latency proportional to the size of the repository, rather than the size of an update.

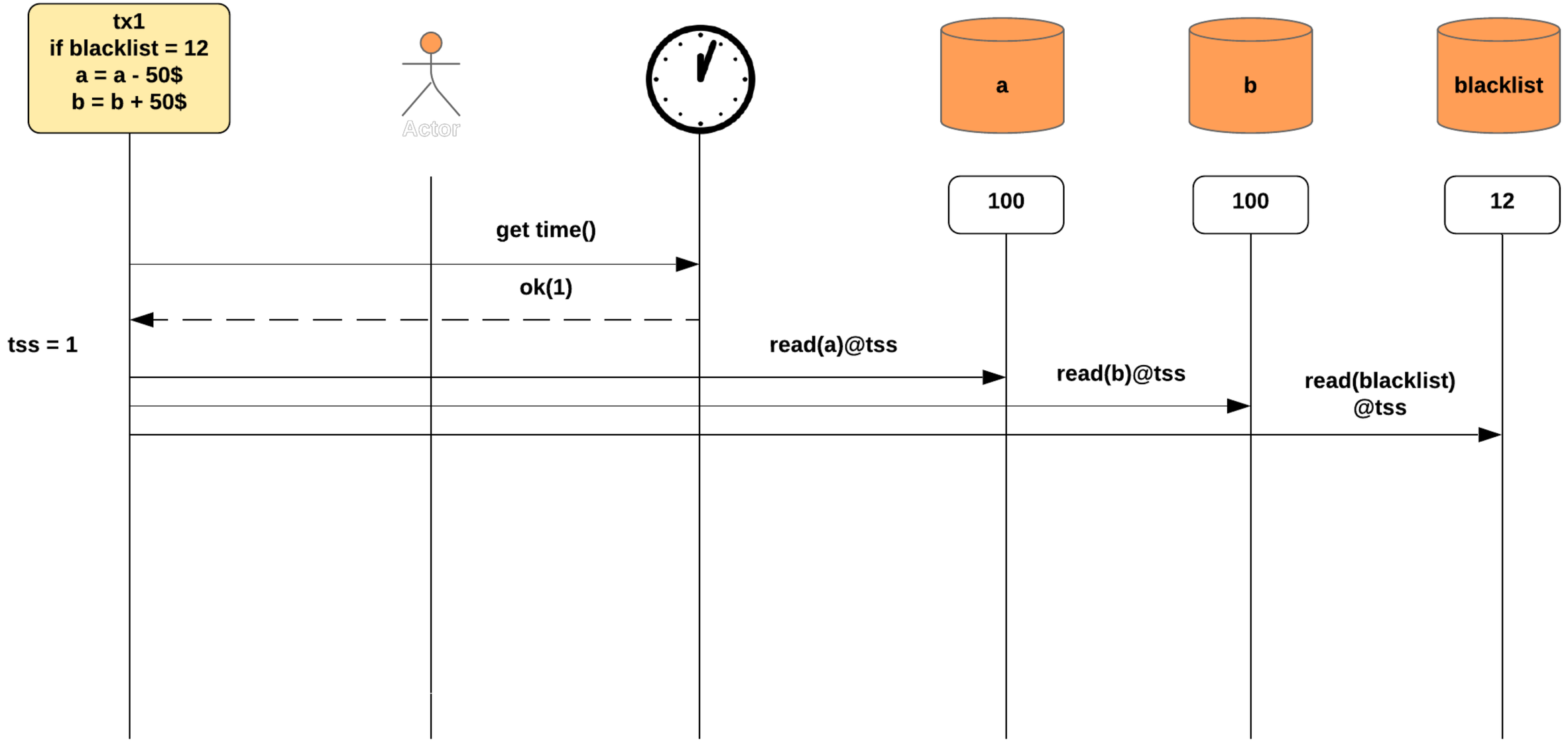
The indexing system could store the repository in a DBMS and update individual documents while using transactions to maintain invariants. However, existing DBMSs can't handle the sheer volume of data: Google's indexing system stores tens of petabytes across thousands of machines [30]. Distributed storage systems like Bigtable [9] can scale to the size of our repository but don't provide tools to help programmers maintain data invariants in the face of concurrent updates.

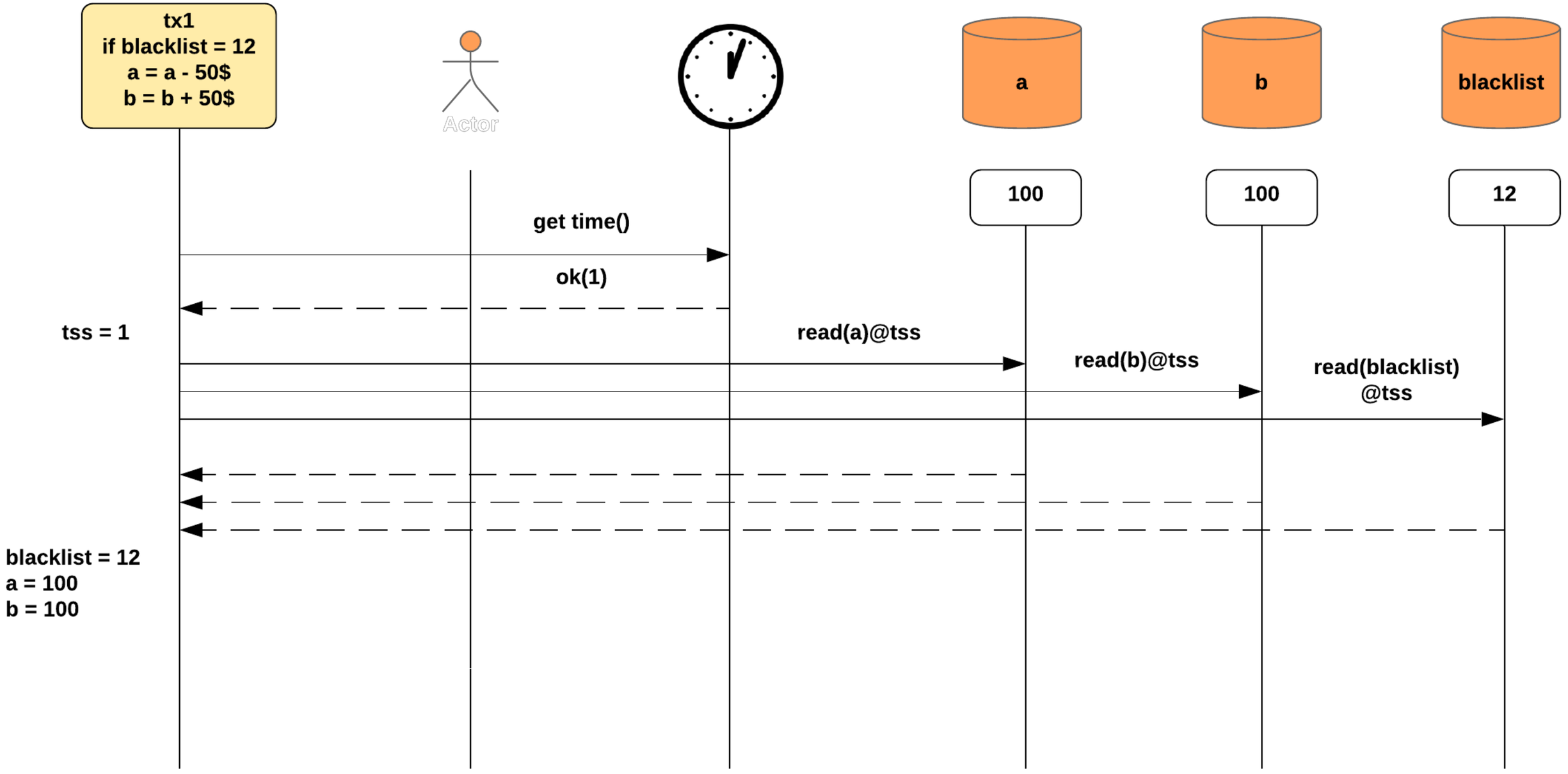
An ideal data processing system for the task of maintaining the web search index would be optimized for *incremental processing*; that is, it would allow us to maintain a very large repository of documents and update it efficiently as each new document was crawled. Given that the system will be processing many small updates concurrently, an ideal system would also provide mechanisms for maintaining invariants despite concurrent updates and for keeping track of which updates have been processed.

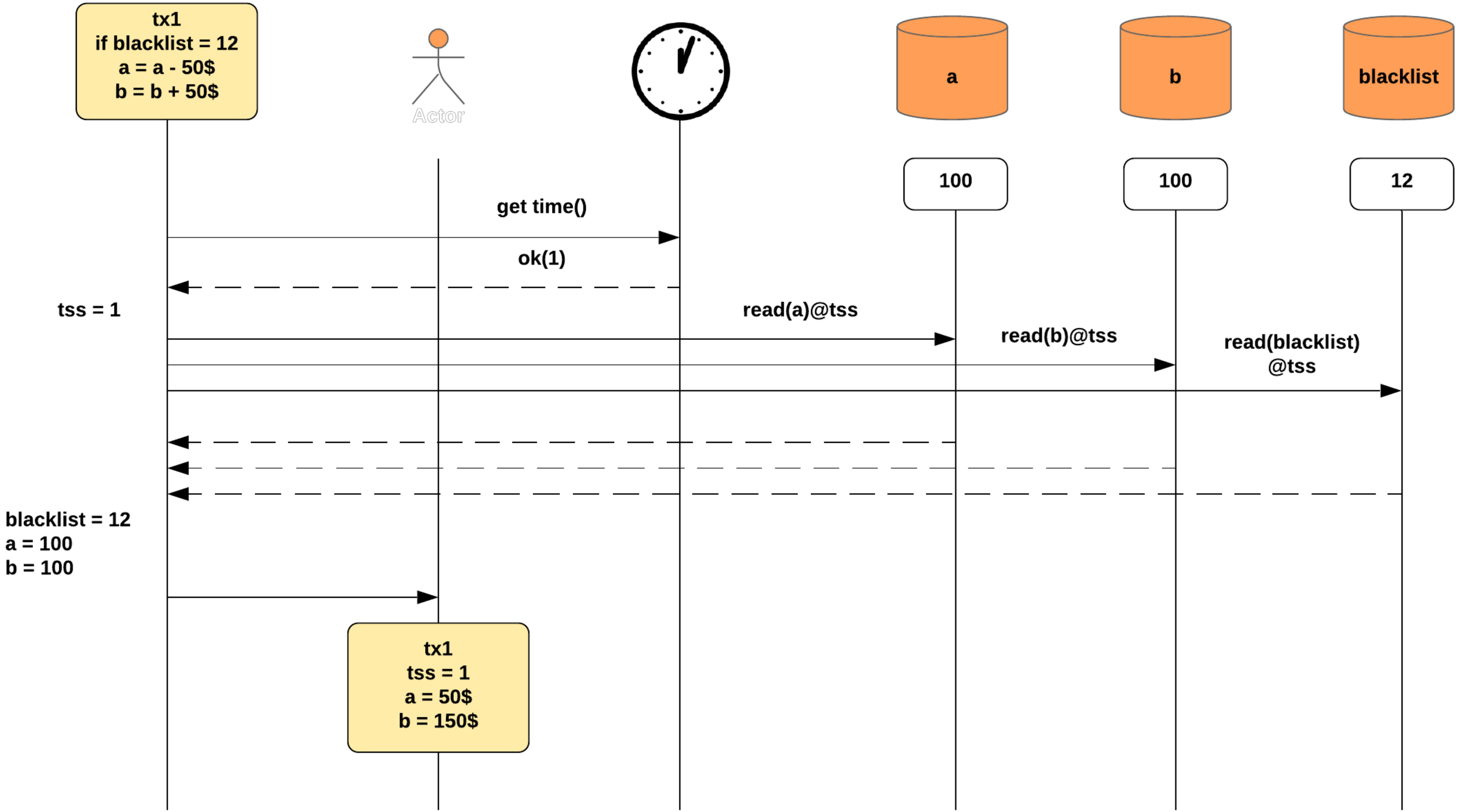
The remainder of this paper describes a particular incremental processing system: Percolator. Percolator provides the user with random access to a multi-PB repository. Random access allows us to process documents in-

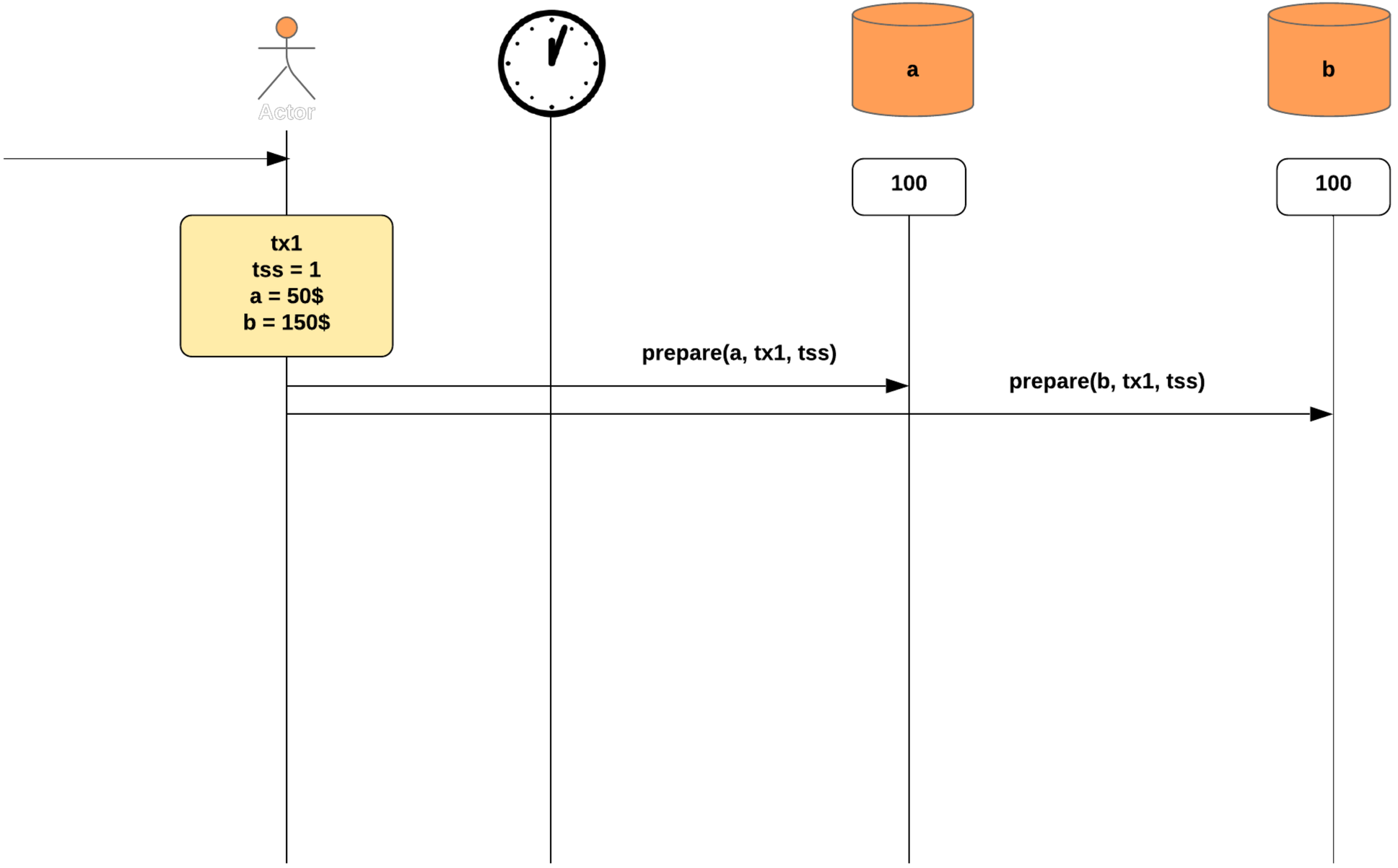

tx1
if blacklist = 12
a = a - 50$
b = b + 50$
Actor
a
b
blacklist
100
100
12

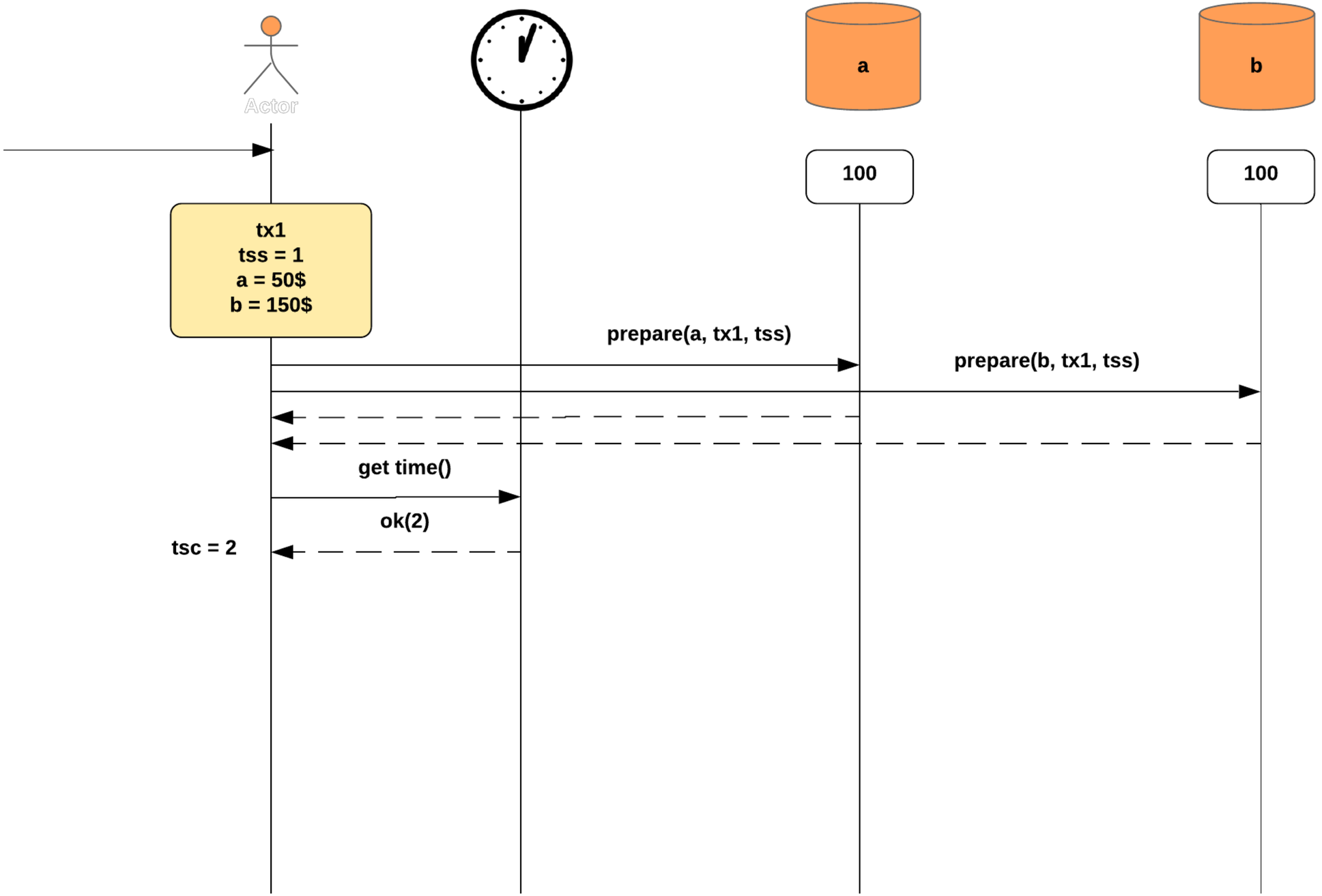

tx1
if blacklist = 12
a = a - 50$
b = b + 50$
Actor
get time()
ok(1)
tss = 1
a
b
blacklist
100
100
12

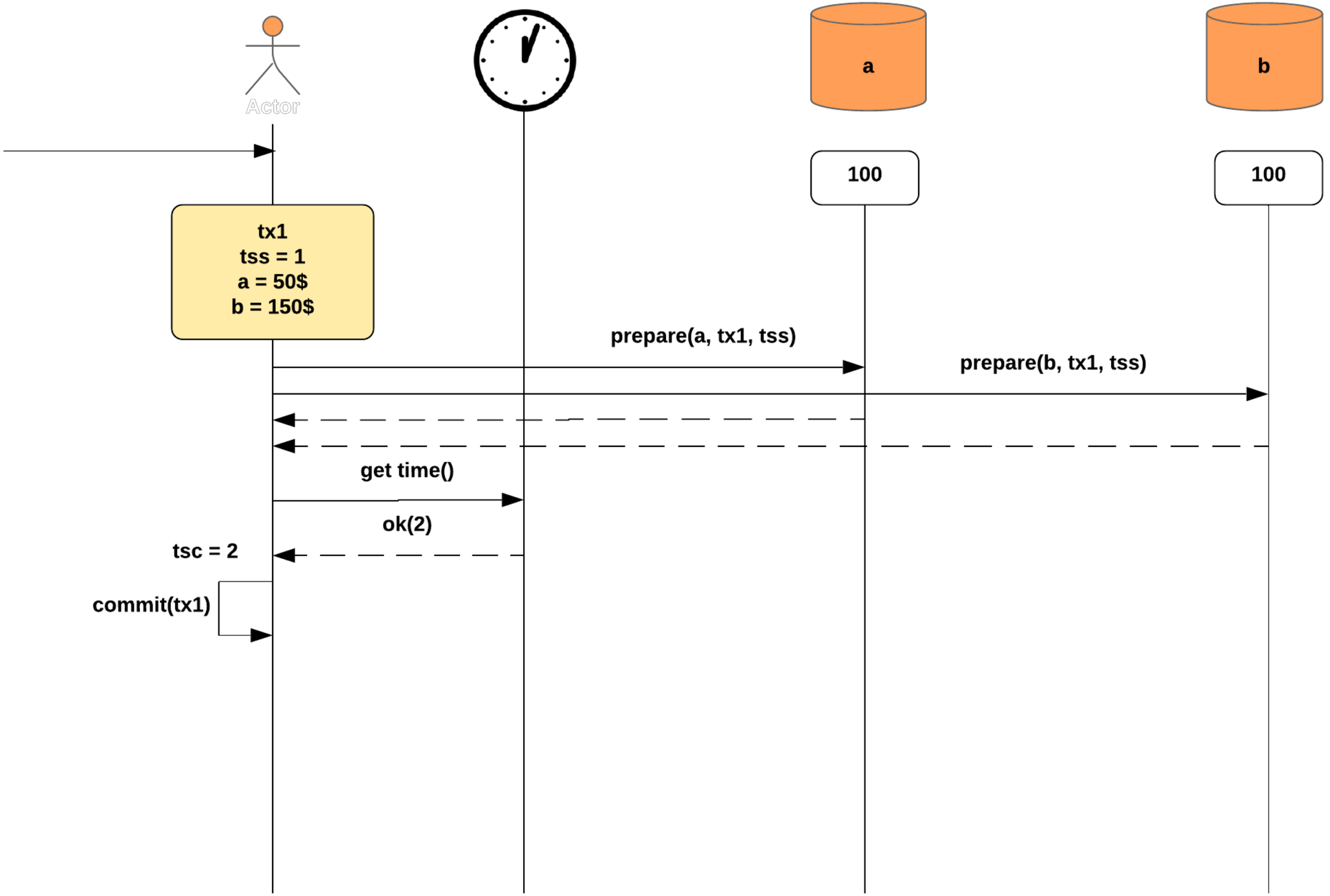

tx1
if blacklist = 12
a = a - 50$
b = b + 50$
Actor
a
b
blacklist
100
100
12
get time()
ok(1)
tss = 1
read(a)@tss
read(b)@tss
read(blacklist)
@tss

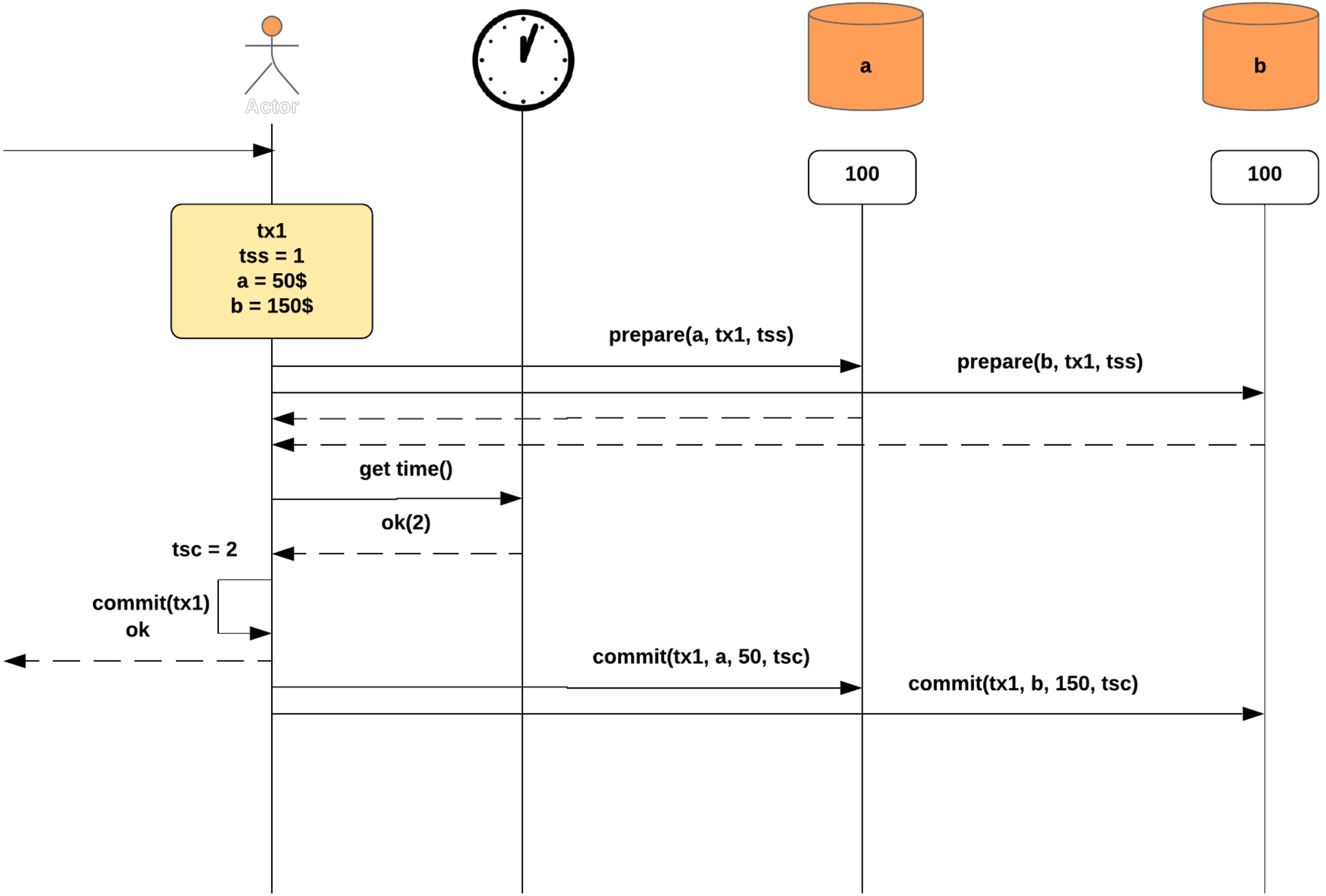

tx1
if blacklist = 12
a = a - 50$
b = b + 50$
Actor
a
b
blacklist
100
100
12
get time()
ok(1)
tss = 1
read(a)@tss
read(b)@tss
read(blacklist)
@tss
blacklist = 12
a = 100
b = 100

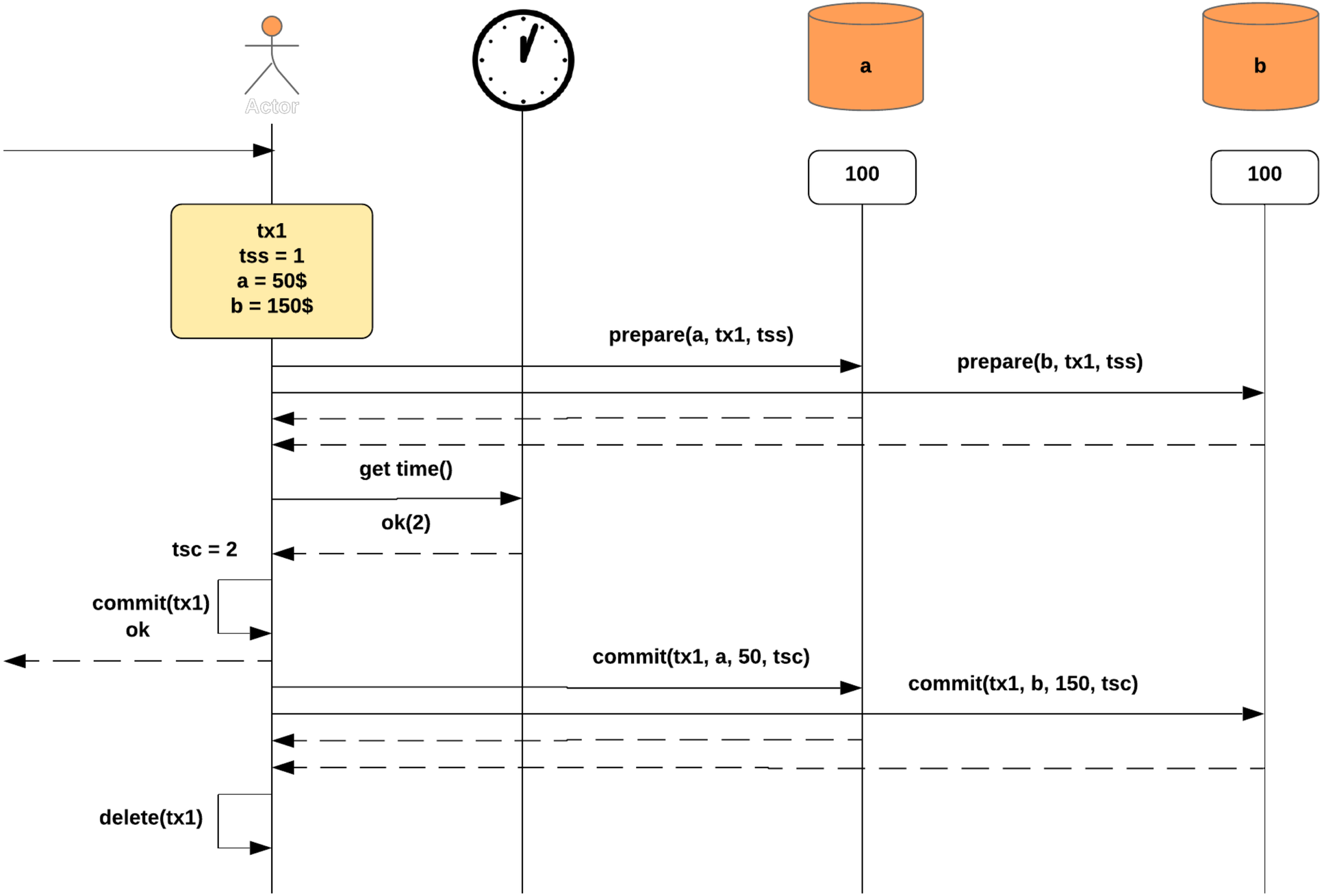

tx1
if blacklist = 12
a = a - 50$
b = b + 50$
Actor
a
b
blacklist
100
100
12
get time()
ok(1)
tss = 1
read(a)@tss
read(b)@tss
read(blacklist)
@tss
blacklist = 12
a = 100
b = 100
tx1
tss = 1
a = 50$
b = 150$


Actor
a
b
100
100
tx1
tss = 1
a = 50$
b = 150$
prepare(a, tx1, tss)
prepare(b, tx1, tss)


Actor
a
b
100
100
tx1
tss = 1
a = 50$
b = 150$
prepare(a, tx1, tss)
prepare(b, tx1, tss)
get time()
ok(2)
tsc = 2


Actor
a
b
100
100
tx1
tss = 1
a = 50$
b = 150$
prepare(a, tx1, tss)
prepare(b, tx1, tss)
get time()
ok(2)
tsc = 2
commit(tx1)

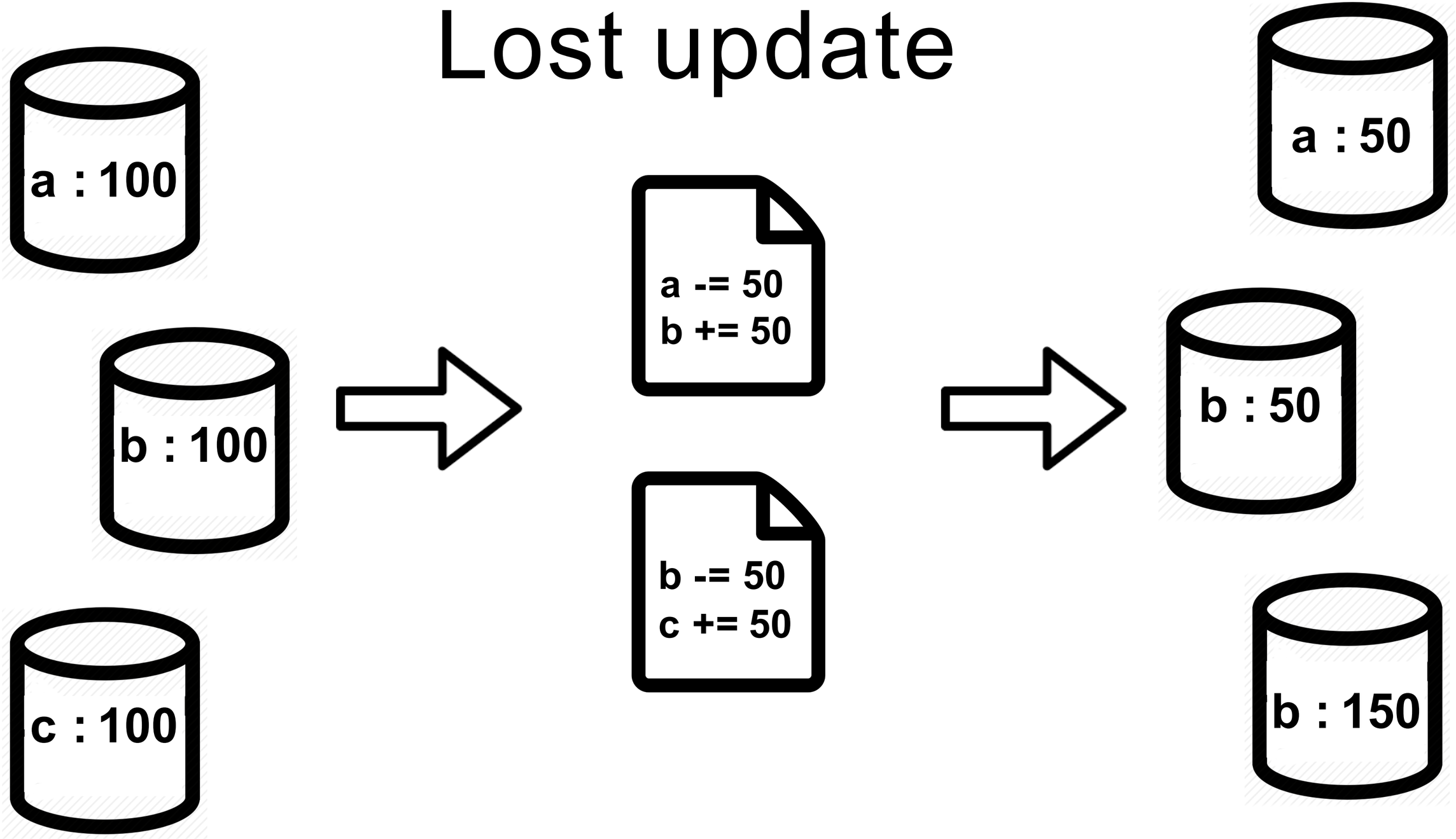

Actor
a
b
100
100
tx1
tss = 1
a = 50$
b = 150$
prepare(a, tx1, tss)
prepare(b, tx1, tss)
get time()
ok(2)
tsc = 2
commit(tx1)
ok
commit(tx1, a, 50, tsc)
commit(tx1, b, 150, tsc)


Actor
a
b
100
100
tx1
tss = 1
a = 50$
b = 150$
prepare(a, tx1, tss)
prepare(b, tx1, tss)
get time()
ok(2)
tsc = 2
commit(tx1)
ok
commit(tx1, a, 50, tsc)
commit(tx1, b, 150, tsc)
delete(tx1)

	with coordinator	clients as coordinators without conflicts	clients as coordinators with 100% conflicts
2PC	81%		
PC + Eiger		990%	1045%
Granola		118%	118%
Percolator	90%	90%	900%

Isolation	Stale read	Write skew	Lost update	Read skew
~~Strict serializability~~				
~~Serializability~~	✓			
~~Snapshot isolation~~	✓	✓		
Read Atomic	✓	✓	✓	
Read Committed	✓	✓	✓	✓

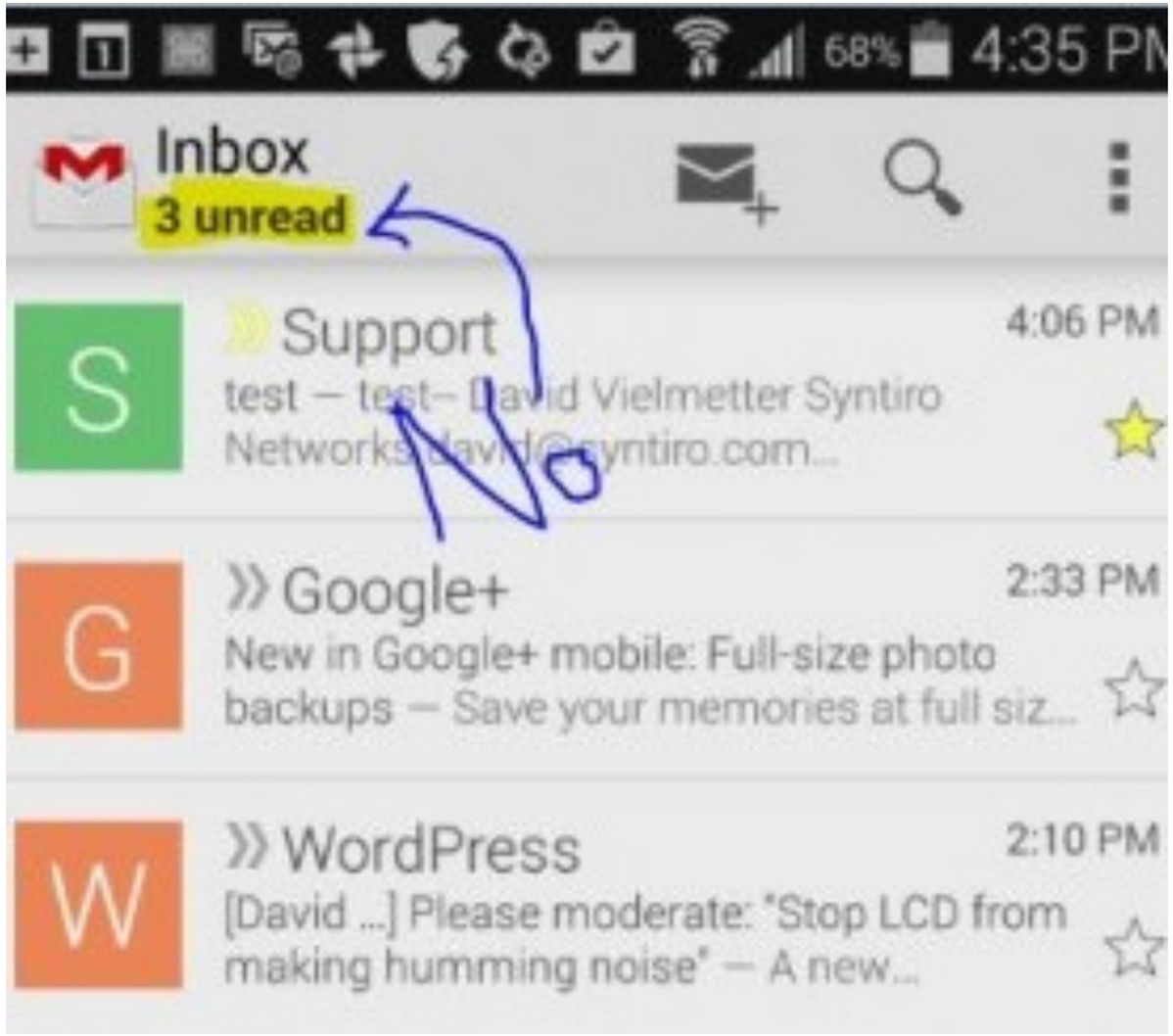
Lost update



Read Atomic

Use cases

- backups
- secondary indexing
- materialized view maintenance



RAMP

Scalable Atomic Visibility with RAMP Transactions

Peter Bailis, Alan Fekete[†], Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and [†]University of Sydney

ABSTRACT

Databases can provide scalability by partitioning data across several servers. However, multi-partition, multi-operation transactional access is often expensive, employing coordination-intensive locking, validation, or scheduling mechanisms. Accordingly, many real-world systems avoid mechanisms that provide useful semantics for multi-partition operations. This leads to incorrect behavior for a large class of applications including secondary indexing, foreign key enforcement, and materialized view maintenance. In this work, we identify a new isolation model—Read Atomic (RA) isolation—that matches the requirements of these use cases by ensuring *atomic visibility*: either all or none of each transaction's updates are observed by other transactions. We present algorithms for Read Atomic Multi-Partition (RAMP) transactions that enforce atomic visibility while offering excellent scalability, guaranteed commit despite partial failures (via *synchronization independence*), and minimized communication between servers (via *partition independence*). These RAMP transactions correctly mediate atomic visibility of updates and provide readers with snapshot access to database state by using limited multi-versioning and by allowing clients to independently resolve non-atomic reads. We demonstrate that, in contrast with existing algorithms, RAMP transactions incur limited overhead—even under high contention—and scale linearly to 100 servers.

1. INTRODUCTION

Faced with growing amounts of data and unprecedented query volume, distributed databases increasingly split their data across multiple servers, or *partitions*, such that no one partition contains an entire copy of the database [7, 13, 18, 19, 22, 29, 43]. This strategy succeeds in allowing near-unlimited scalability for operations that access single partitions. However, operations that access multiple partitions must communicate across servers—often synchronously—in order to provide correct behavior. Designing systems and algorithms that tolerate these communication delays is a difficult task but is key to maintaining scalability [17, 28, 29, 35].

In this work, we address a largely underserved class of applications requiring multi-partition, atomically visible[1] *transactional* access: cases where all or none of each transaction's effects should be visible. The status quo for these multi-partition atomic transactions provides an uncomfortable choice between algorithms that are fast but deliver inconsistent results and algorithms that deliver consistent results but are often slow and unavailable under failure. Many of the largest modern, real-world systems opt for protocols that guarantee fast and scalable operation but provide few—if any—transactional semantics for operations on arbitrary sets of data items [11, 13, 15, 22, 26, 38, 44]. This results in incorrect behavior for use cases that require atomic visibility, including secondary indexing, foreign key constraint enforcement, and materialized view maintenance (Section 2). In contrast, many traditional transactional mechanisms correctly ensure atomicity of updates [8, 17, 43]. However, these algorithms—such as two-phase locking and variants of optimistic concurrency control—are often coordination-intensive, slow, and, under failure, unavailable in a distributed environment [5, 18, 28, 35]. This dichotomy between scalability and atomic visibility has been described as "a fact of life in the big cruel world of huge systems" [25]. The proliferation of non-transactional multi-item operations is symptomatic of a widespread "fear of synchronization" at scale [9].

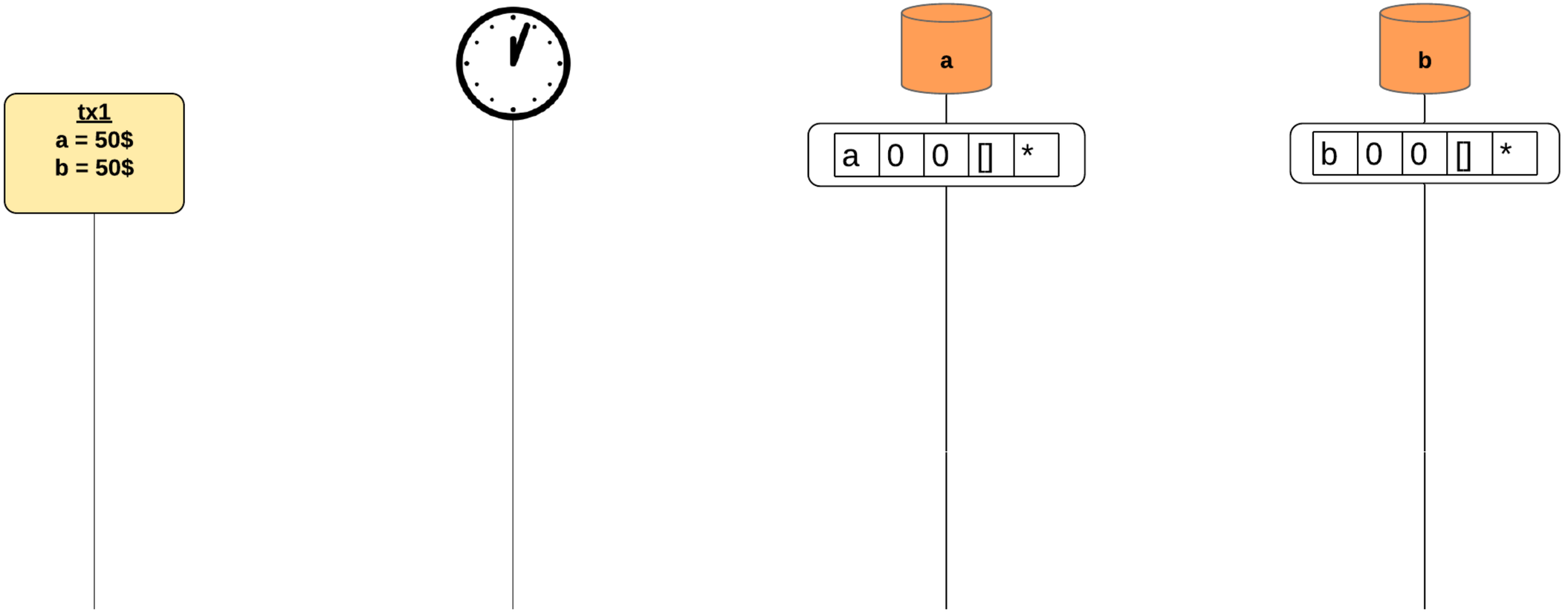
Our contribution in this paper is to demonstrate that atomically visible transactions on partitioned databases are *not* at odds with scalability. Specifically, we provide high-performance implementations of a new, non-serializable isolation model called Read Atomic (RA) isolation. RA ensures that all or none of each transaction's updates are visible to others and that each transaction reads from an atomic snapshot of database state (Section 3)—this is useful in the applications we target. We subsequently develop three new, scalable algorithms for achieving RA isolation that we collectively title Read Atomic Multi-Partition (RAMP) transactions (Section 4). RAMP transactions guarantee scalability and outperform existing atomic algorithms because they satisfy two key scalability constraints. First, RAMP transactions guarantee *synchronization independence*: one client's transactions cannot cause another client's transactions to stall or fail. Second, RAMP transactions guarantee *partition independence*: clients never need to contact partitions that their transactions do not directly reference. Together, these properties ensure guaranteed completion, limited coordination across partitions, and horizontal scalability for multi-partition access.

RAMP transactions are scalable because they appropriately control the visibility of updates without inhibiting concurrency. Rather than force concurrent reads and writes to stall, RAMP transactions allow reads to "race" writes: RAMP transactions can autonomously detect the presence of non-atomic (partial) reads and, if necessary, repair them via a second round of communication with servers. To accomplish this, RAMP writers attach metadata to each write and use limited multi-versioning to prevent readers from stalling. The three algorithms we present offer a trade-off between the size of this metadata and performance. `RAMP-Small` transactions require constant space (a timestamp per write) and two round trip time delays (RTTs) for reads and writes. `RAMP-Fast` transactions require metadata size that is linear in the number of writes in the transaction but only require one RTT for reads in the common case and two in the worst case. `RAMP-Hybrid` transactions employ Bloom filters [10] to provide an intermediate solution. Traditional techniques like locking

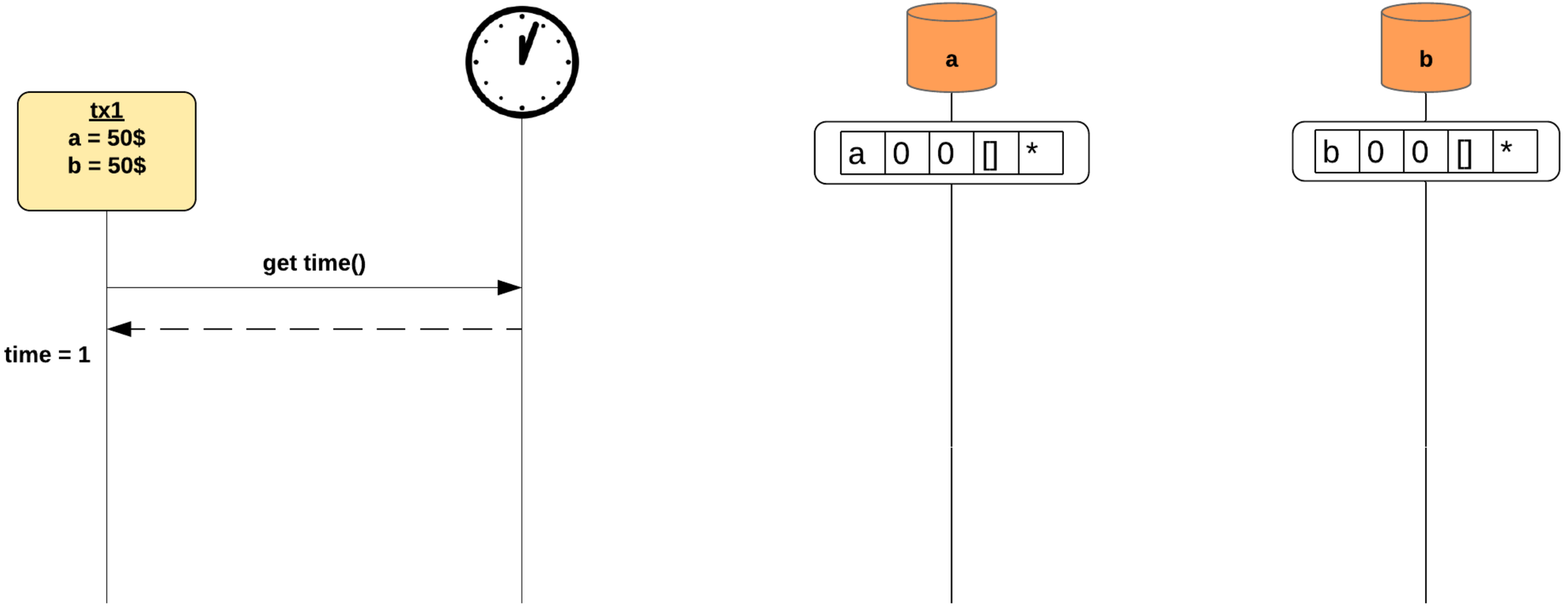
[1]Our use of "atomic" (specifically, Read Atomic isolation) concerns all-or-nothing *visibility* of updates (i.e., the ACID isolation effects of ACID atomicity; Section 3). This differs from uses of "atomicity" to denote serializability [8] or linearizability [4].

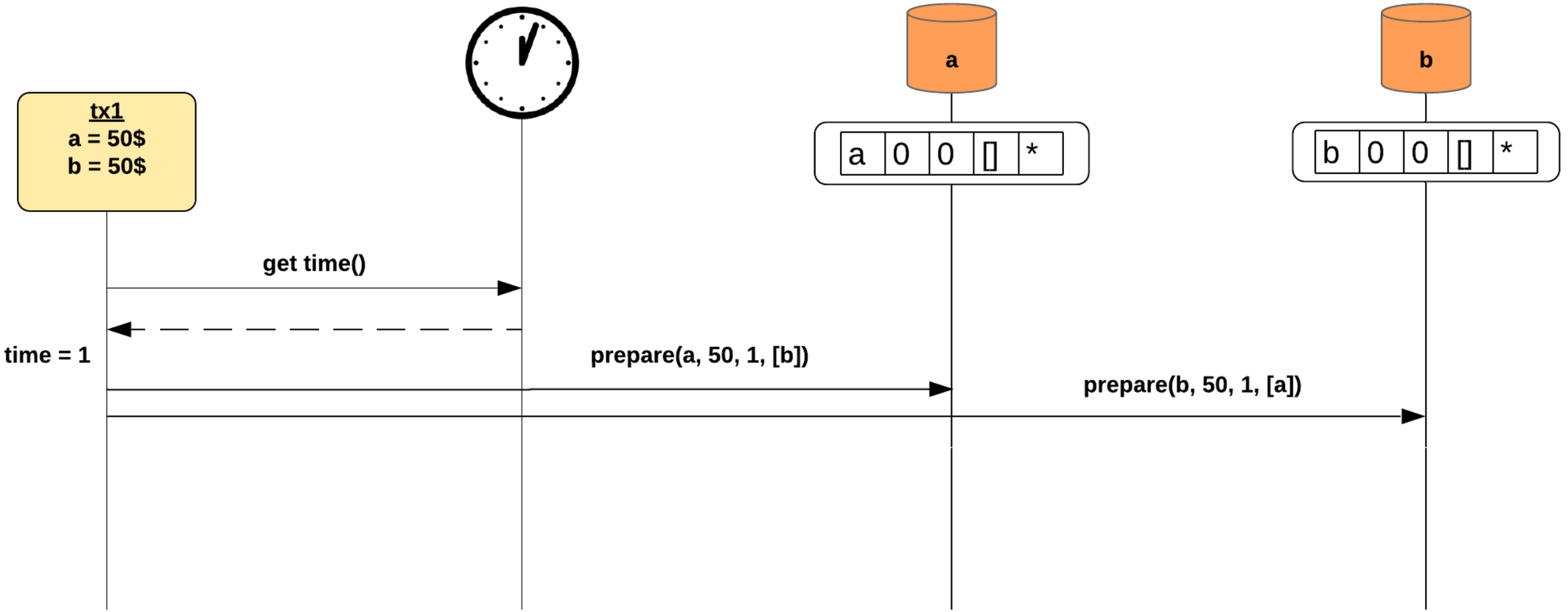
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

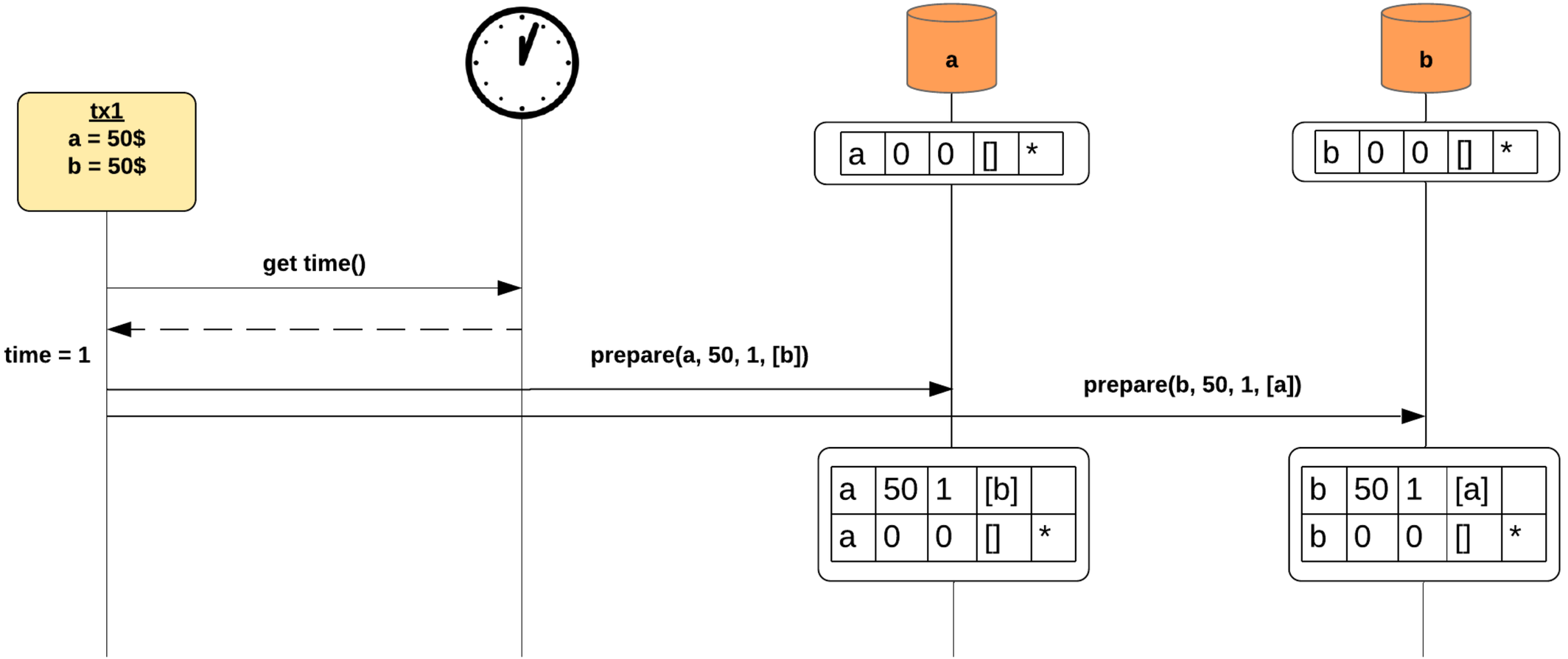
ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2588562.

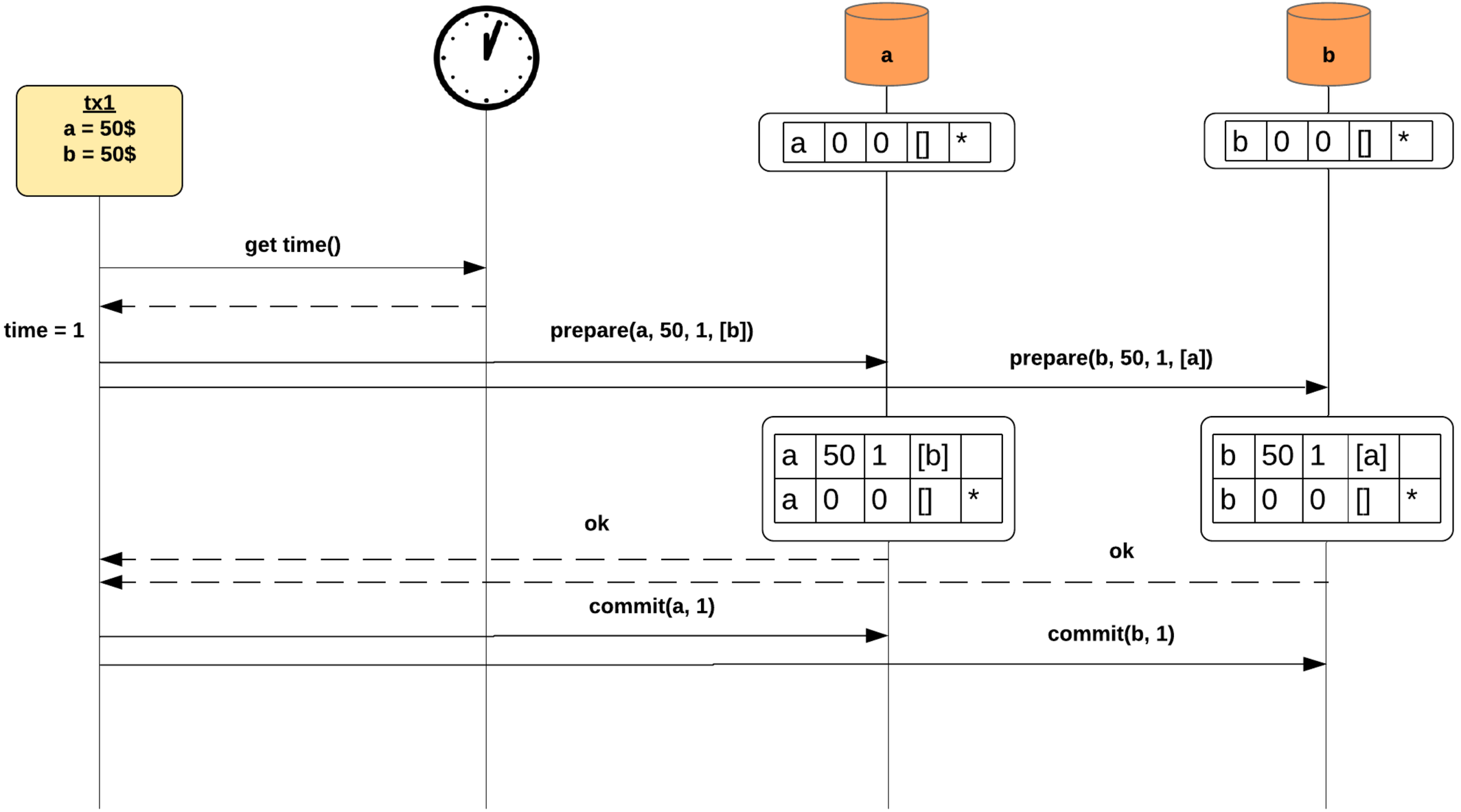


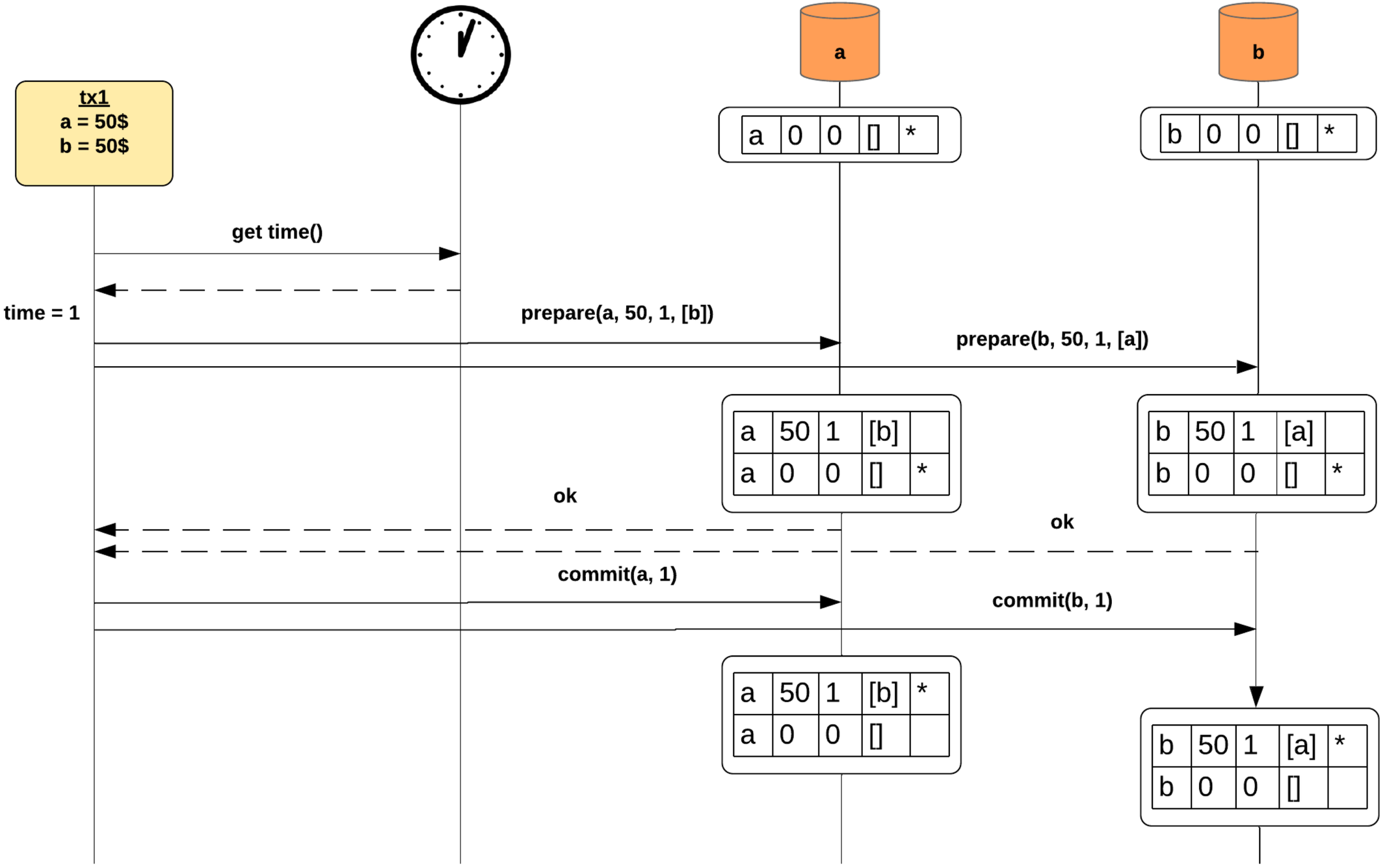
How to write

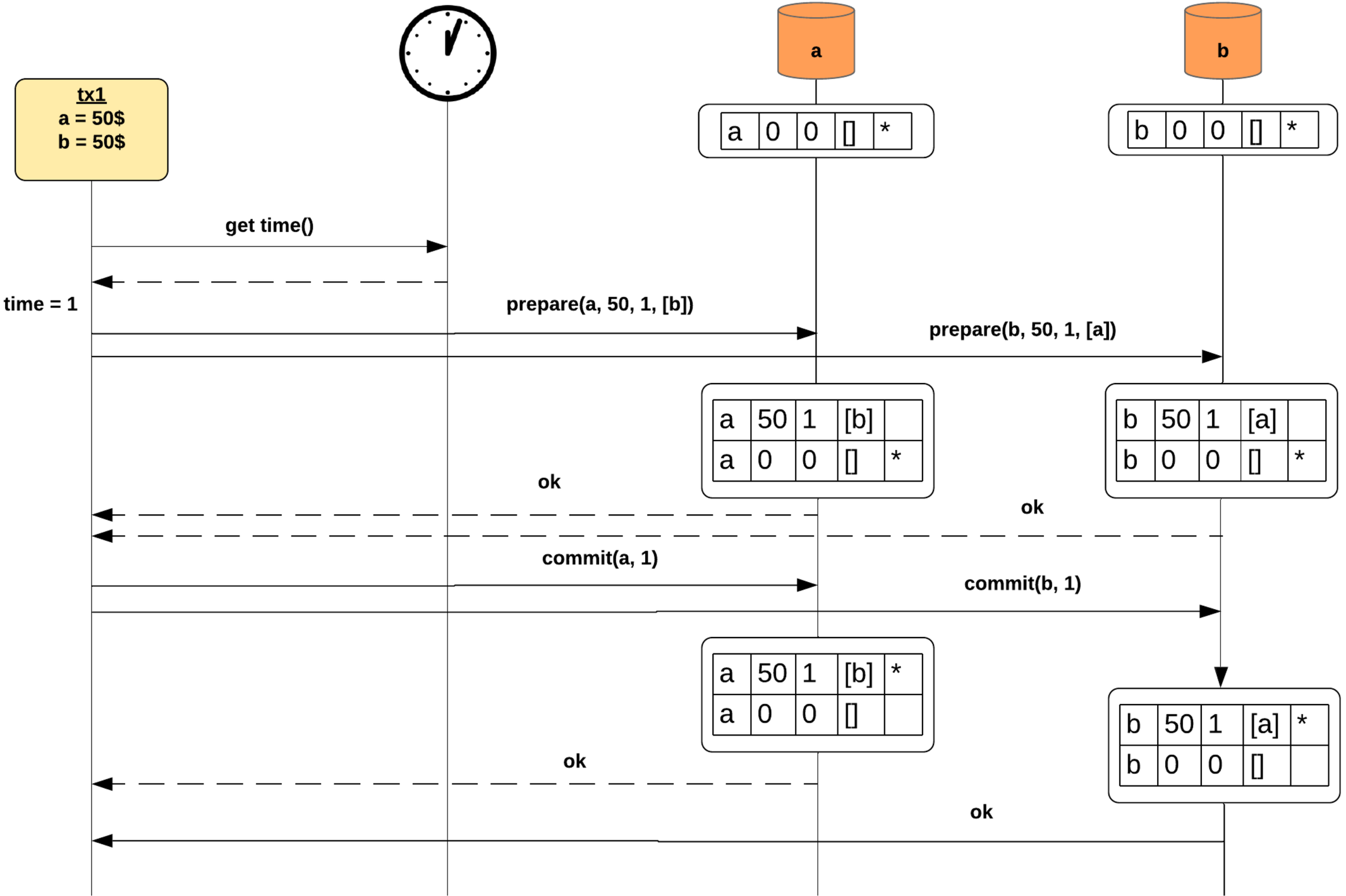

tx1
a = 50$
b = 50$
get time()
time = 1
a
a 0 0 [] *
b
b 0 0 [] *

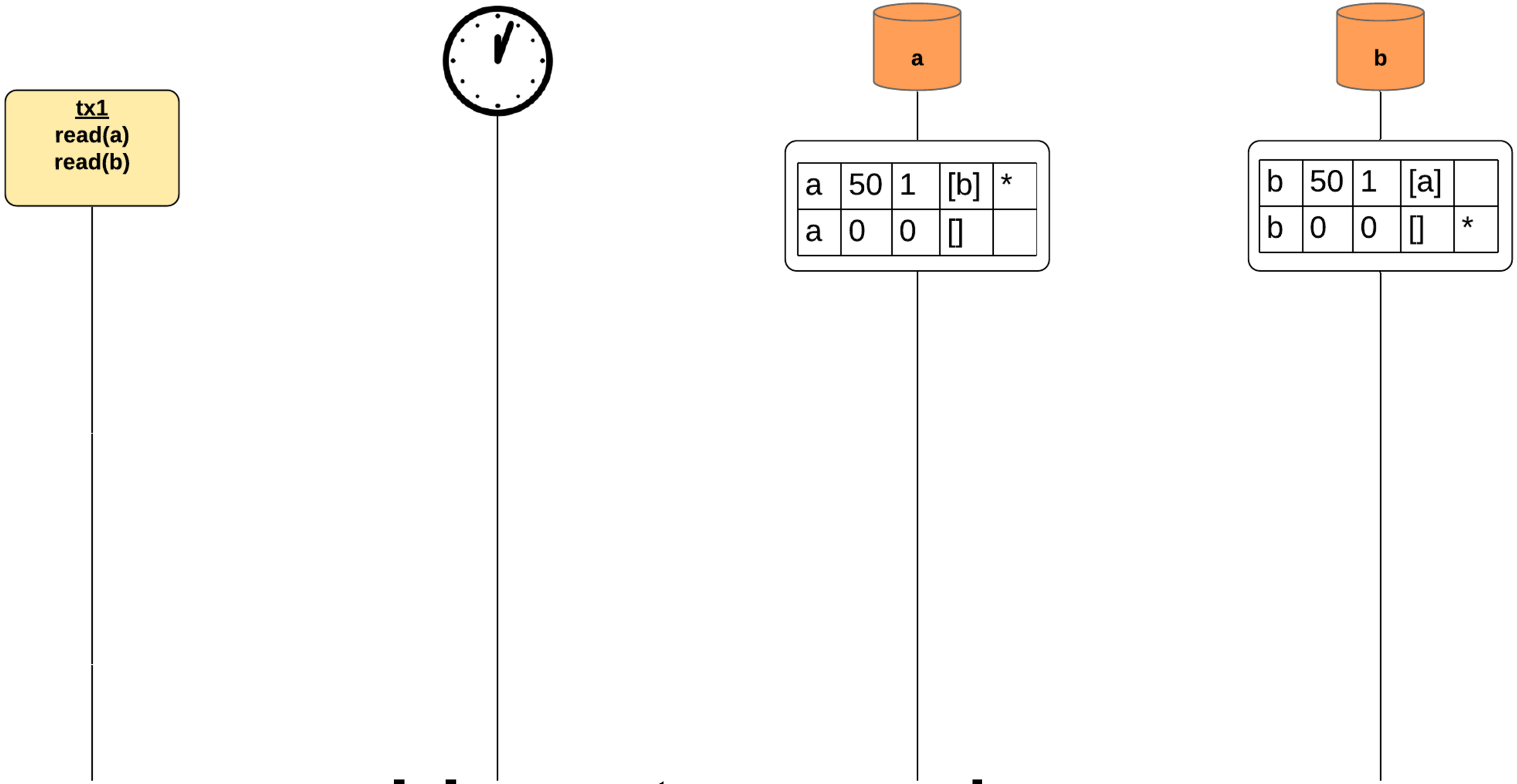

tx1
a = 50$
b = 50$
a
b
a 0 0 [] *
b 0 0 [] *
get time()
time = 1
prepare(a, 50, 1, [b])
prepare(b, 50, 1, [a])


tx1
a = 50$
b = 50$
a
b
a 0 0 [] *
b 0 0 [] *
get time()
time = 1
prepare(a, 50, 1, [b])
prepare(b, 50, 1, [a])
a 50 1 [b]
a 0 0 [] *
b 50 1 [a]
b 0 0 [] *

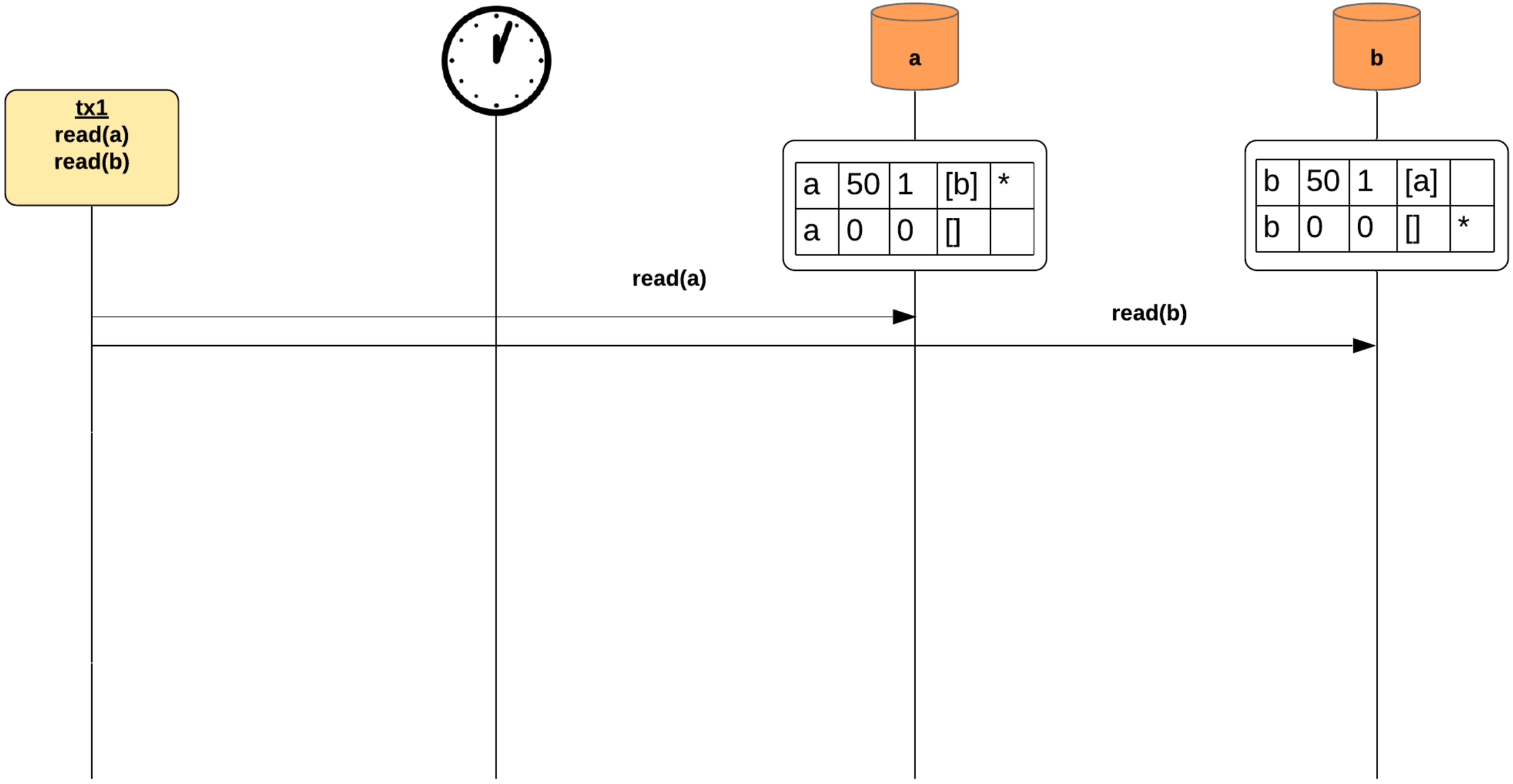

tx1
a = 50$
b = 50$
a
b
a 0 0 [] *
b 0 0 [] *
get time()
time = 1
prepare(a, 50, 1, [b])
prepare(b, 50, 1, [a])
a 50 1 [b]
a 0 0 [] *
b 50 1 [a]
b 0 0 [] *
ok
ok
commit(a, 1)
commit(b, 1)

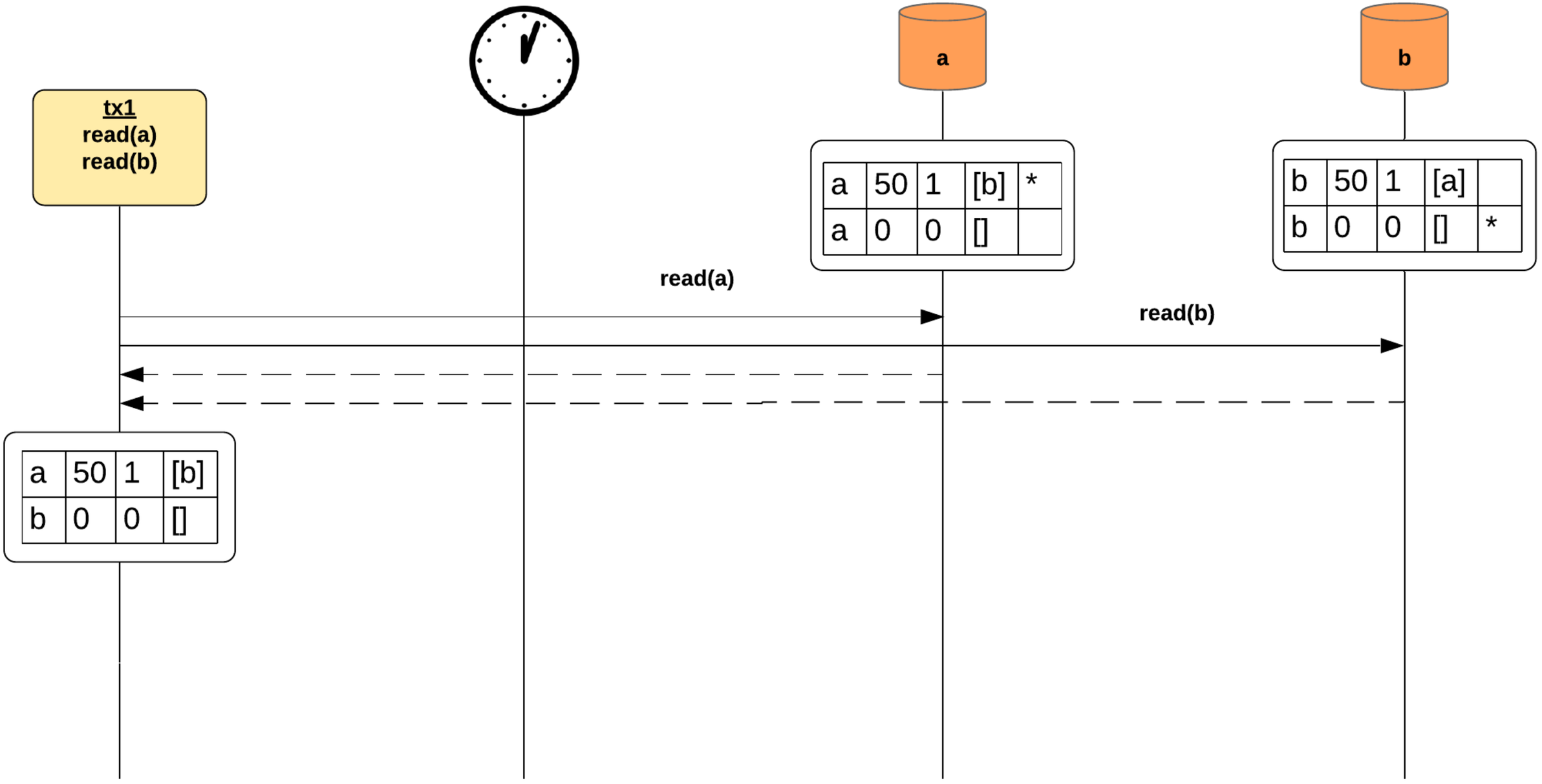

tx1
a = 50$
b = 50$
a
b
a 0 0 [] *
b 0 0 [] *
get time()
time = 1
prepare(a, 50, 1, [b])
prepare(b, 50, 1, [a])
a 50 1 [b]
a 0 0 [] *
b 50 1 [a]
b 0 0 [] *
ok
ok
commit(a, 1)
commit(b, 1)
a 50 1 [b] *
a 0 0 []
b 50 1 [a] *
b 0 0 []

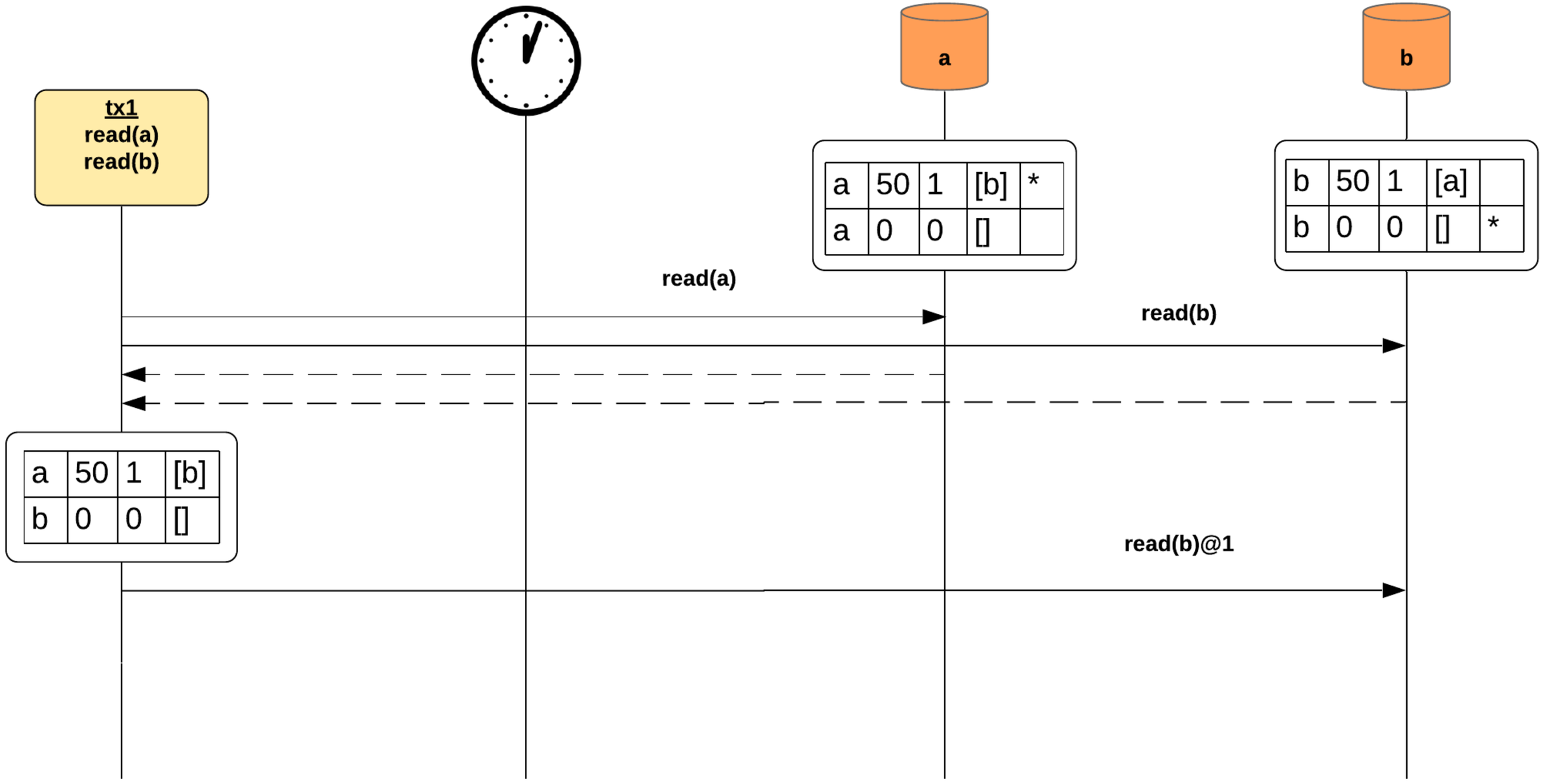

tx1
a = 50$
b = 50$
a
b
a 0 0 [] *
b 0 0 [] *
get time()
time = 1
prepare(a, 50, 1, [b])
prepare(b, 50, 1, [a])
a 50 1 [b]
a 0 0 [] *
b 50 1 [a]
b 0 0 [] *
ok
ok
commit(a, 1)
commit(b, 1)
a 50 1 [b] *
a 0 0 []
b 50 1 [a] *
b 0 0 []
ok
ok

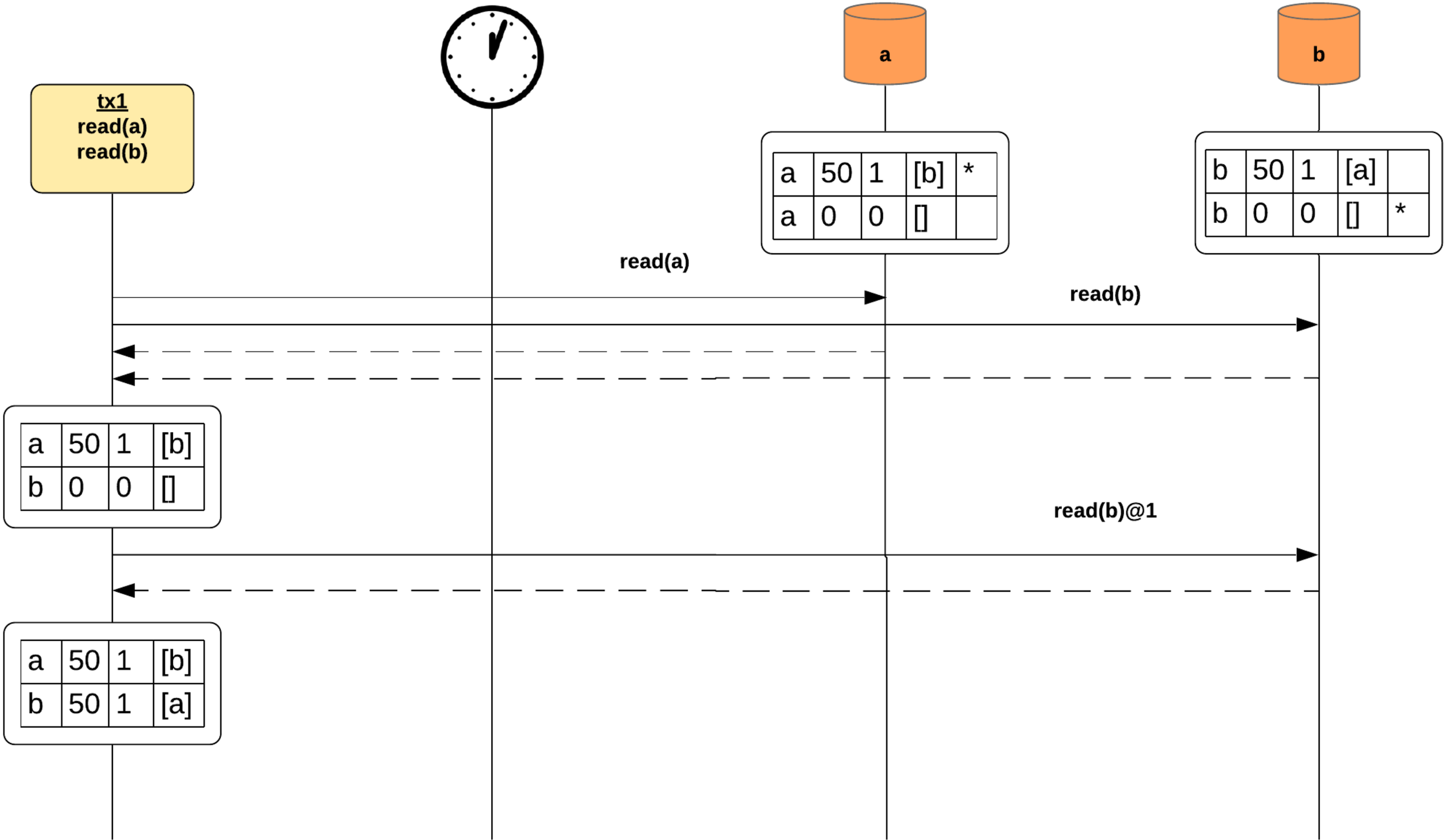


How to read


tx1
read(a)
read(b)
a
b
a 50 1 [b] *
a 0 0 []
b 50 1 [a]
b 0 0 [] *
read(a)
read(b)


tx1
read(a)
read(b)
a
b
a 50 1 [b] *
a 0 0 []
b 50 1 [a]
b 0 0 [] *
read(a)
read(b)
a 50 1 [b]
b 0 0 []


tx1
read(a)
read(b)
a
b
a 50 1 [b] *
a 0 0 []
b 50 1 [a]
b 0 0 [] *
read(a)
read(b)
a 50 1 [b]
b 0 0 []
read(b)@1


tx1
read(a)
read(b)
a
b
a 50 1 [b] *
a 0 0 []
b 50 1 [a]
b 0 0 [] *
read(a)
read(b)
a 50 1 [b]
b 0 0 []
read(b)@1
a 50 1 [b]
b 50 1 [a]

	with coordinator	clients as coordinators without conflicts	clients as coordinators with 100% conflicts
2PC	55%		
PC	88%	788%	4322%
PC + Eiger		111%	555%
Granola		77%	77%
RAMP		66%	66%

Key takeaway

- new protocols were invented since 2000s

Key takeaway

- new protocols were invented since 2000s
- distributed transactions became practical

Key takeaway

- new protocols were invented since 2000s
- distributed transactions became practical
- the cost (work / latency) overhead is 70-120% depending on a protocol and workload

Key takeaway

- new protocols were invented since 2000s
- distributed transactions became practical
- the cost (work / latency) overhead is 70-120% depending on a protocol and workload
- read papers, make experiment

- https://bitcointalk.org/index.php?topic=499580
- https://twitter.com/bradfitz/status/885288352244576256
- https://m.habr.com/ru/post/258449/
- Life beyond TX: https://queue.acm.org/detail.cfm?id=3025012
- https://www.oracle.com/technetwork/products/clustering/overview/distributed-transactions-and-xa-163941.pdf
- https://en.wikipedia.org/wiki/CAP_theorem
- SNOW: https://www.usenix.org/system/files/conference/osdi16/osdi16-lu.pdf
- Cosmos DB: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction
- Paxos Commit: https://lamport.azurewebsites.net/video/consensus-on-transaction-commit.pdf
- Eiger: https://www.cs.cmu.edu/~dga/papers/eiger-nsdi2013.pdf
- Granola: https://www.usenix.org/system/files/conference/atc12/atc12-final118.pdf
- Percolator: https://ai.google/research/pubs/pub36726
- RAMP: http://www.bailis.org/papers/ramp-sigmod2014.pdf

Спасибо!

Denis Rystsov
email: derystso@microsoft.com
twitter: @rystsov