Consistency in Transactional Distributed Databases: Protocols and Testing

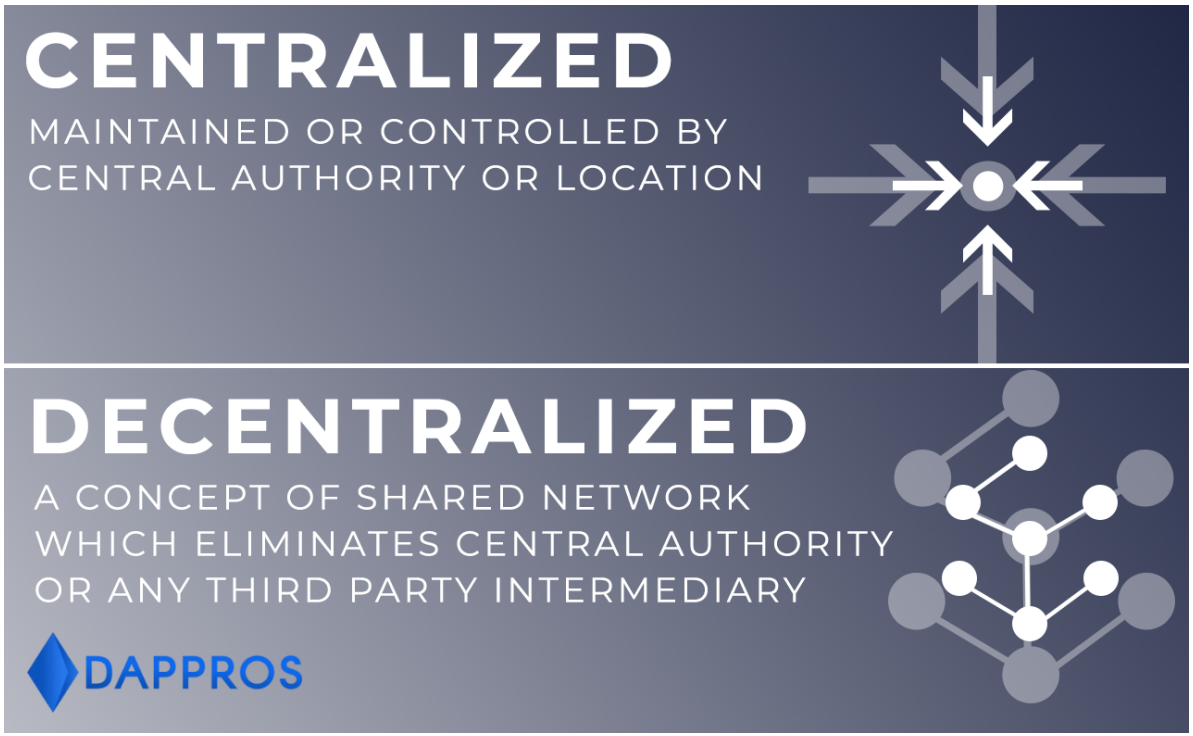
Hengfeng Wei (魏恒峰)

hfwei@nju.edu.cn

Nov. 04, 2022

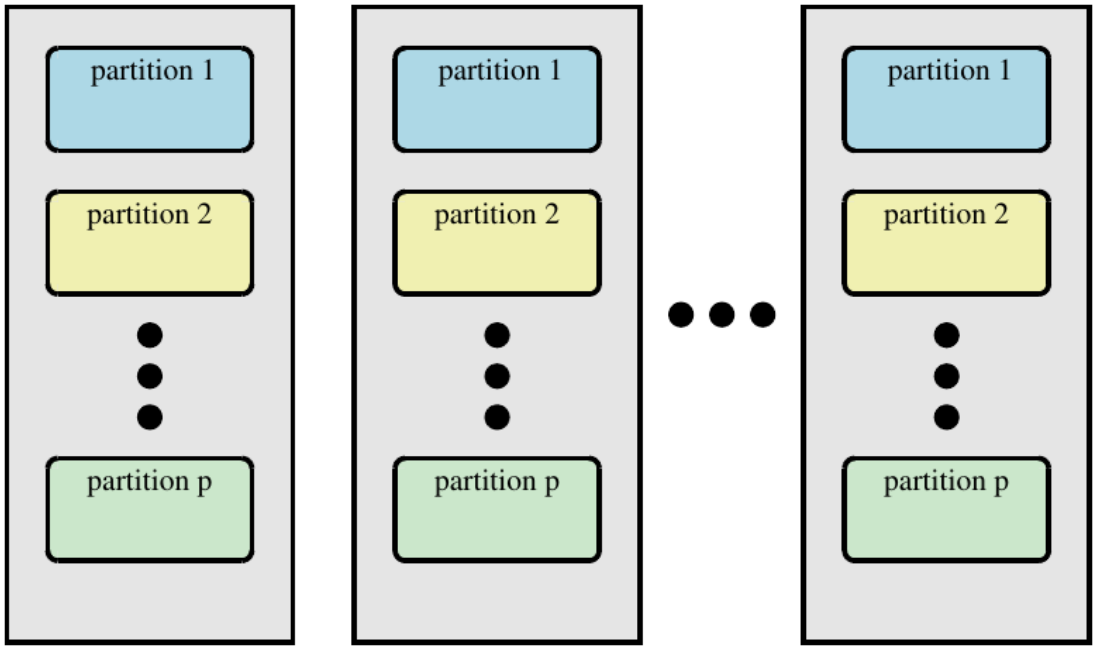
Background

Centralized Databases *vs.* Distributed Databases



Data Consistency Problem

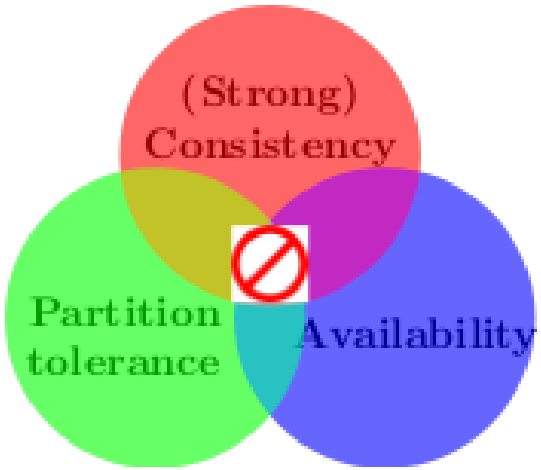
"Data Partition + Data Replication"



Data Consistency Problem

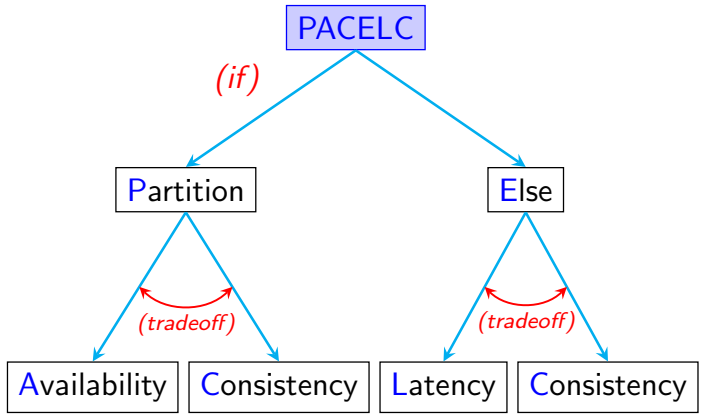
Data Consistency Problem

(Strong) Consistency, Availability, Latency、Patition tolerance



CAP Theorem

(Brewer@PODC2000)

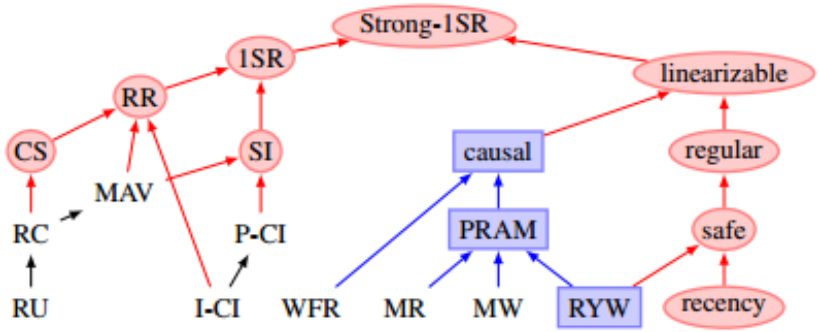


PACELC Tradeoff

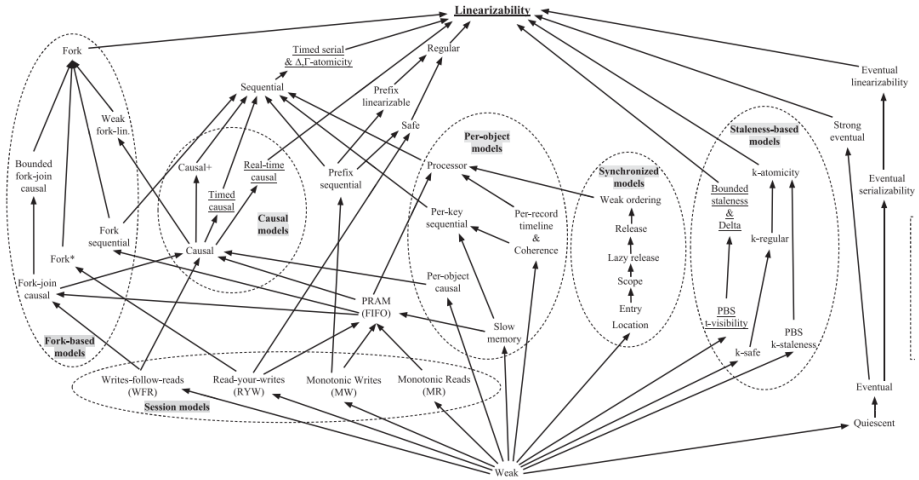
(Abadi@Computer2012)

Consistency Models

Use consistency models to capture these tradeoffs



(Bailis@VLDB2012)



(Viotti@CSUR2016)

My Researches

Researches on data consistency around consistency models:

Computability: What is possible or impossible?

Protocol: How to design fast, scalable, and fault-tolerant protocols?

Testing: What is the complexity?

How to design efficient testing algorihthms?

My Researches

Researches on data consistency around consistency models:

Computability: What is possible or impossible?

Protocol: How to design fast, scalable, and fault-tolerant protocols?

Testing: What is the complexity?

How to design efficient testing algorihthms?

Classic problems with the ever-changing requirements

My Researches (I)

Read/Write Register[a] ($\geq$ 2012)

Distributed Non-transactional Key-Value Stores

[a]读写寄存器，也就是读写变量，虽然最初与计算机系统中的“寄存器”概念相关，但已慢慢解耦。

My Researches

Wei:TPDS2016

Wei:TC2017

Wei:PODC2020-Huang

My Researches (II)

Replicated Data Types[b] ($\geq$ 2017)

(a) Google Docs

(b) Apache Wave

(c) Wikipedia

(d) LaTeX Editor

[b] 复制数据类型，是经典数据类型的分布式版本，如列表，集合，队列等。

My Researches (II)

Wei:PODC-BA2018

Wei:OPODIS2018

Wei:SRDS2020-Jiang

Wei:JOS2020-Ji

My Researches (III)

Distributed Transactions[c] ($\geq$ 2020)

[c] 分布式事务。每个事务由一组操作构成，这些操作要么全成功，要么全不成功。

My Researches (III)

Wei:ATC2021

Overview of the Work on UniStore

UniStore: A fault-tolerant marriage of causal and strong consistency

Manuel Bravo Alexey Gotsman Borja de Régil Hengfeng Wei *
IMDEA Software Institute *Nanjing University*

ATC'2021 (CCF A)

UniStore is the first fault-tolerant and scalable transactional data store that combines causal and strong consistency.

Overview of the Work on UniStore

Partial Order-Restrictions (PoR) Consistency

$$\text{CC} < \text{PoR} < \text{SER}$$

CC: CausalConsistency; SER: Serializability

Key Challenges (I): Ensure liveness in presence of faults

Key Challenges (II): Provide rigorous correctness proof

Overview of the Work on UniStore

UniStore: A fault-tolerant marriage of causal and strong consistency

Manuel Bravo Alexey Gotsman Borja de Régil Hengfeng Wei *
IMDEA Software Institute *Nanjing University*

ATC'2021 (CCF A)

I am fully responsible for the rigorous correctness proof:

- Finished a proof of 20 pages contained in the arXiv version
- Identified several nontrivial bugs in the early versions of the protocol[d]

[d] One of these bugs also exists in the well-known Granola protocol proposed by James Cowling and Barbara Liskov, something that had gone unnoticed for 10 years.

What is UniStore?

UniStore is a fast, scalable, and fault-tolerant

transactional distributed key-value store

that supports a combination of weak and strong consistency.

What is UniStore?

UniStore is a fast, scalable, and fault-tolerant

transactional distributed key-value store

that supports a combination of weak and strong consistency.

Weak consistency: CausalConsistency

Strong consistency: Serializability

Why Uni-?

Weak consistency: low latency, high availability,
but unable to preserve critical application invariants

Strong consistency: easy to preserve critical application invariants,
but require global synchronization

The Uni- Approach

- Allow multiple consistency levels to coexist
- Take the weak consistency level as the default and the baseline
- Allow programmers to choose the operations that should be executed under strong consistency
  - e.g., if the execution of an operation may violate the application violations

The Uni- Approach

Partial Order-Restrictions (PoR) Consistency

(Li@OSDI'2012, Li@ACT'2018)

- PoR allows programmers to classify operations as either causal or strong.

The Uni- Approach

Partial Order-Restrictions (PoR) Consistency

(Li@OSDI'2012, Li@ACT'2018)

- PoR allows programmers to classify operations as either causal or strong.
- Causal operations satisfy CausalConsistency:
  - Clients see updates in an order that respects the potential causality between them.
  - Causally independent operations can be executed concurrently.

The Uni- Approach

Partial Order-Restrictions (PoR) Consistency

(Li@OSDI'2012, Li@ACT'2018)

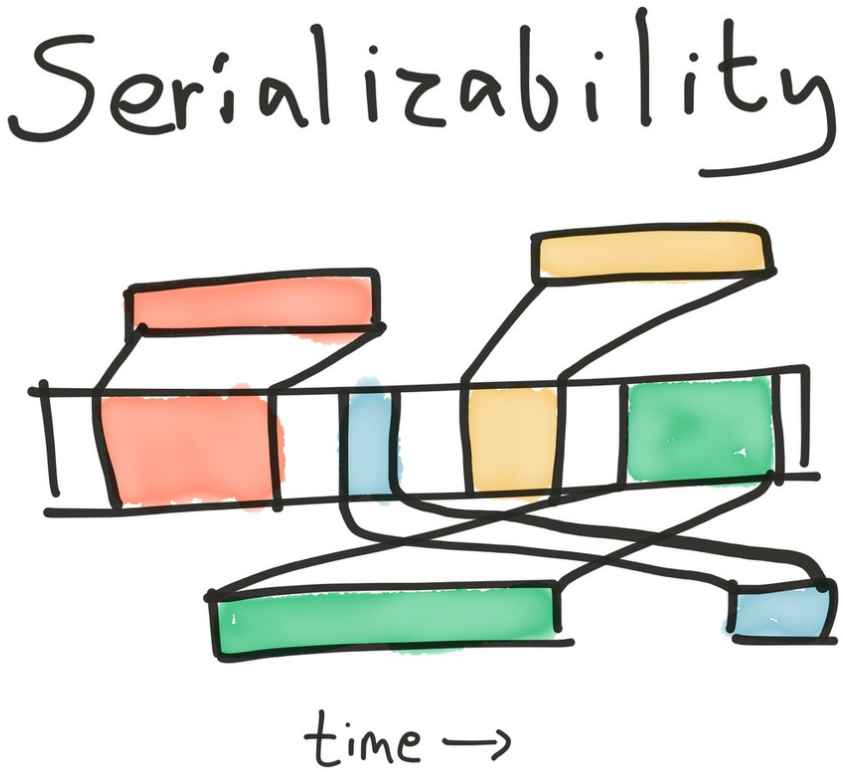
- PoR allows programmers to classify operations as either causal or strong.
- Causal operations satisfy CausalConsistency:
  - Clients see updates in an order that respects the potential causality between them.
  - Causally independent operations can be executed concurrently.
- Programmers use strong operations to enforce orders between some causally independent operations.

Serializability and CausalConsistency

TODO: +fig Clients see updates in an order that respects the potential causality between them.

Causally independent operations can be executed concurrently.

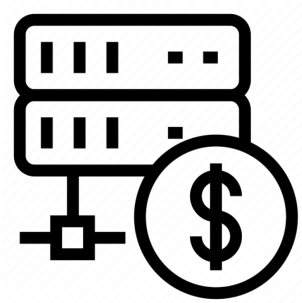
Serializability and CausalConsistency

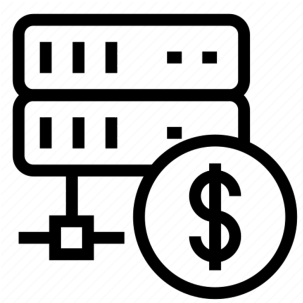


All the operations seem to be executed in some sequential order.

A Banking Application

DEPOSIT WITHDRAW QUERY INTEREST

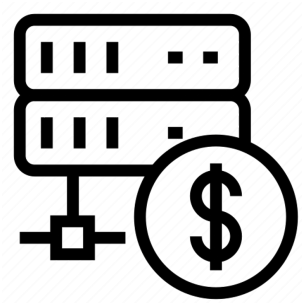


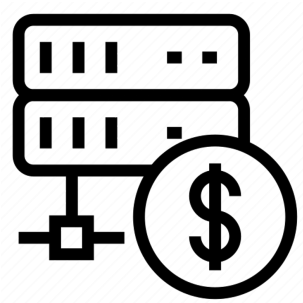


Invariant: balance ≥ 0

A Banking Application

DEPOSIT WITHDRAW QUERY INTEREST



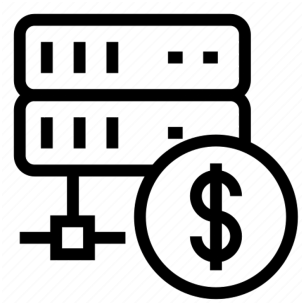


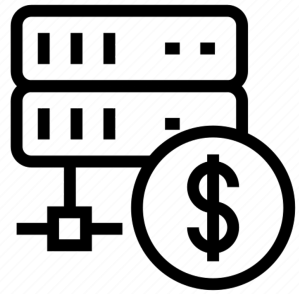
Invariant: balance ≥ 0

DEPOSIT operations can be executed under CAUSALCONSISTENCY.

A Banking Application

DEPOSIT WITHDRAW QUERY INTEREST





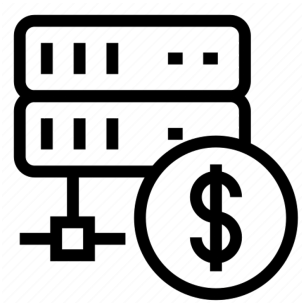
Invariant: balance ≥ 0

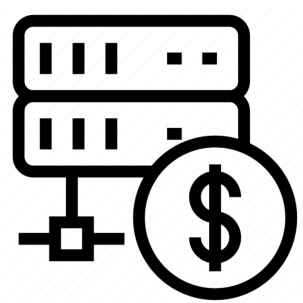
DEPOSIT operations can be executed under CAUSALCONSISTENCY.

DEPOSIT(50)

A Banking Application

DEPOSIT WITHDRAW QUERY INTEREST

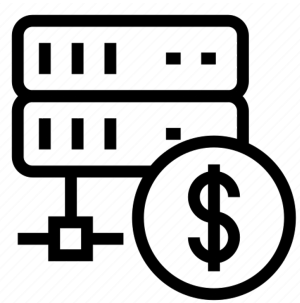


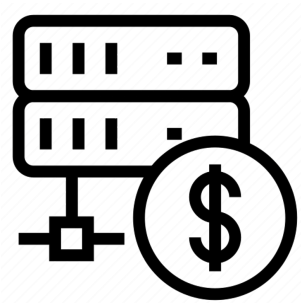


Invariant: balance ≥ 0

A Banking Application

DEPOSIT WITHDRAW QUERY INTEREST



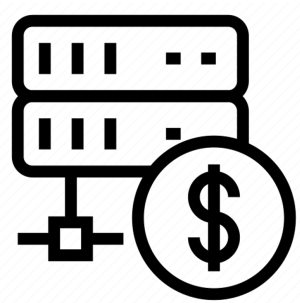


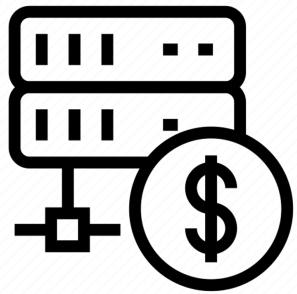
Invariant: balance ≥ 0

Causal consistency also allows two causally independent WITHDRAW to execute concurrently, without knowing each other.

A Banking Application

DEPOSIT WITHDRAW QUERY INTEREST





Invariant: balance ≥ 0

Causal consistency also allows two causally independent WITHDRAW to execute concurrently, without knowing each other.

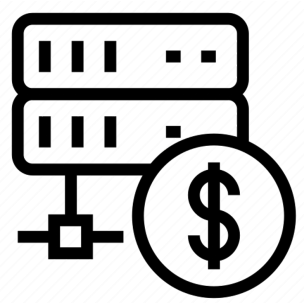
WITHDRAW(60)

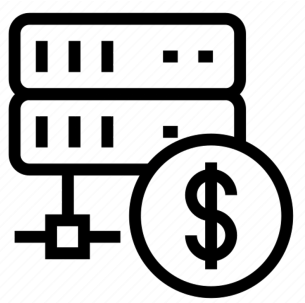
The PoR Approach

- The programmer provides a symmetric conflict relation $\bowtie$ on operations.
- Any operation involved in the conflict relation is marked strong.
- PoR ensures that conflicting operations are executed serially.

A Banking Application

DEPOSIT WITHDRAW QUERY INTEREST





Invariant: balance ≥ 0

Only WITHDRAW are marked strong.
Declaring that strong transactions
including WITHDRAW on the same account conflict.

Consistency Model of UniStore(Safety)

UniStore implements a transactional variant of the PoR consistency

Consistency Model of UniStore(Safety)

UniStore implements a transactional variant of the PoR consistency

$$T \triangleq T_{causal} \uplus T_{strong}$$

Consistency Model of UniStore(Safety)

UniStore implements a transactional variant of the PoR consistency

$$T \triangleq T_{causal} \uplus T_{strong}$$

(I) transactional causal consistency by default

(II) to specify conflicting transactions under strong consistency

Consistency Model of UniStore(Liveness)

EventualVisibility

A transaction $t \in T$ that is either strong or originates at a correct data center eventually become visible at all correct data centers.

- from some point on, t precedes all transactions issued at correct data centers.

Design Challenges of UniStore

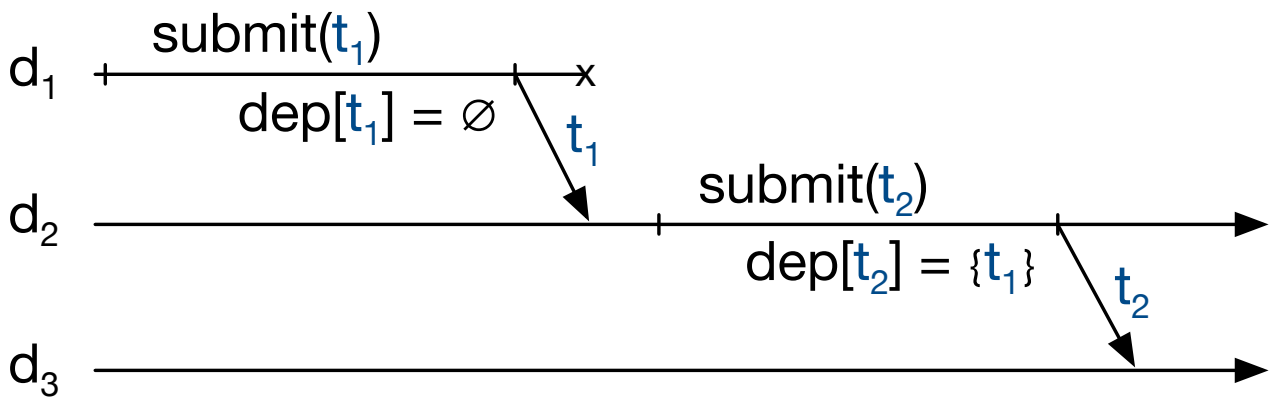
How to satisfy liveness (EventualVisibility) despite failures?

A transaction $t \in T$ that is either strong or originates at a correct data center eventually become visible at all correct data centers.

Liveness of Causal Transactions

Data center d_1 crashes

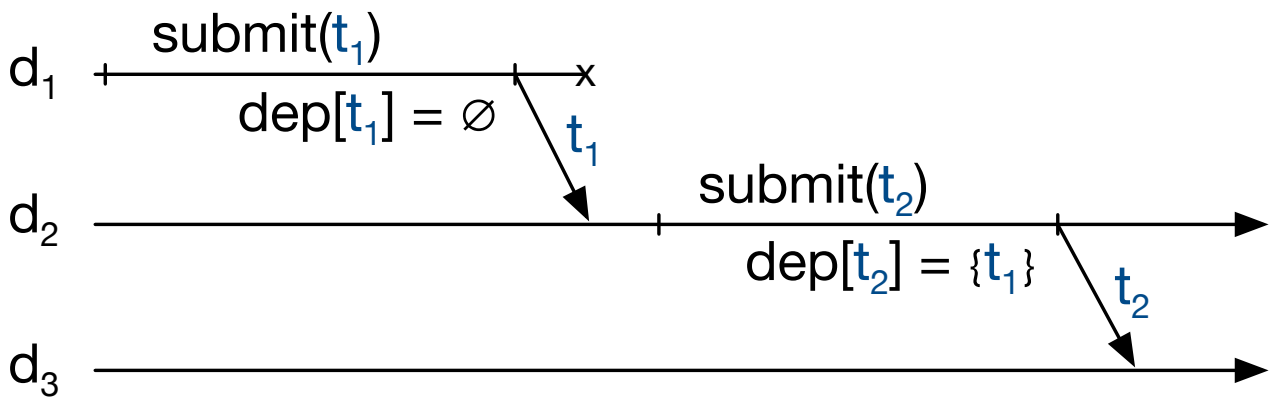
before t_1 is replicated to correct data center d_3.



Liveness of Causal Transactions

Data center d_1 crashes

before t_1 is replicated to correct data center d_3.



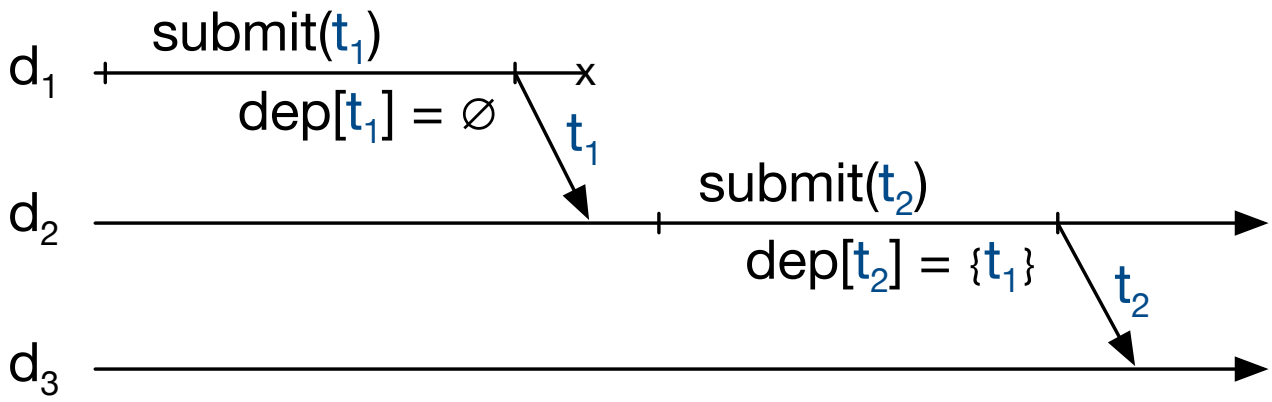
Transaction t_2 (at correct data center d_2)

may never become visible at correct data center d_3.

Liveness of Causal Transactions

Data center d_1 crashes

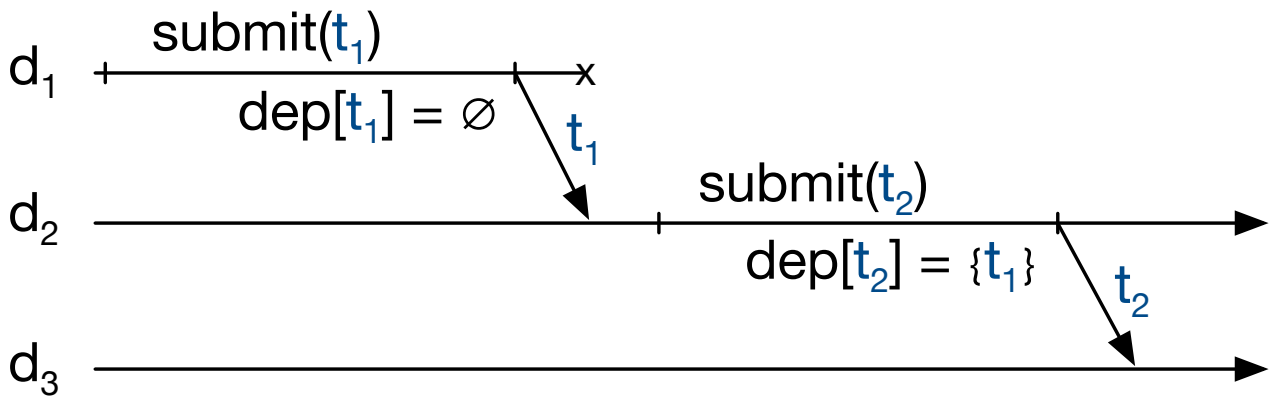
before t_1 is replicated to correct data center d_3.



Liveness of Causal Transactions

Data center d_1 crashes

before t_1 is replicated to correct data center d_3.



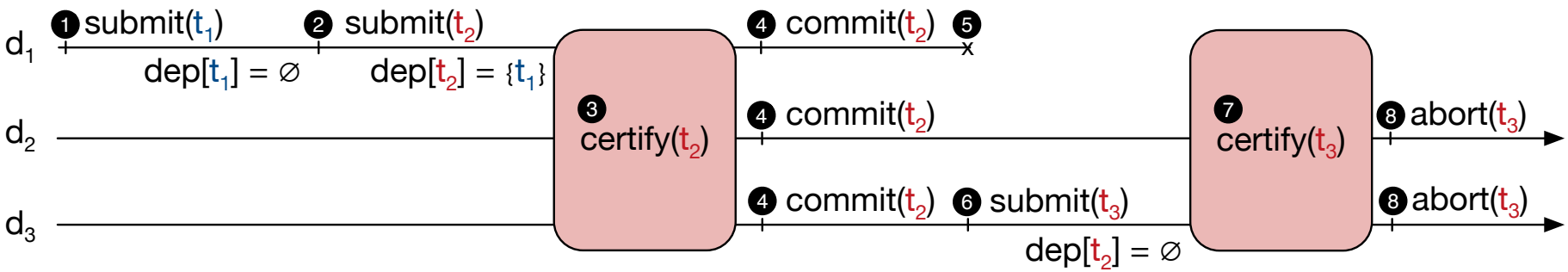
Data center d_2 need to forward causal transactions

to other data centers.

Liveness of Strong Transactions

Data center d_1 crashes

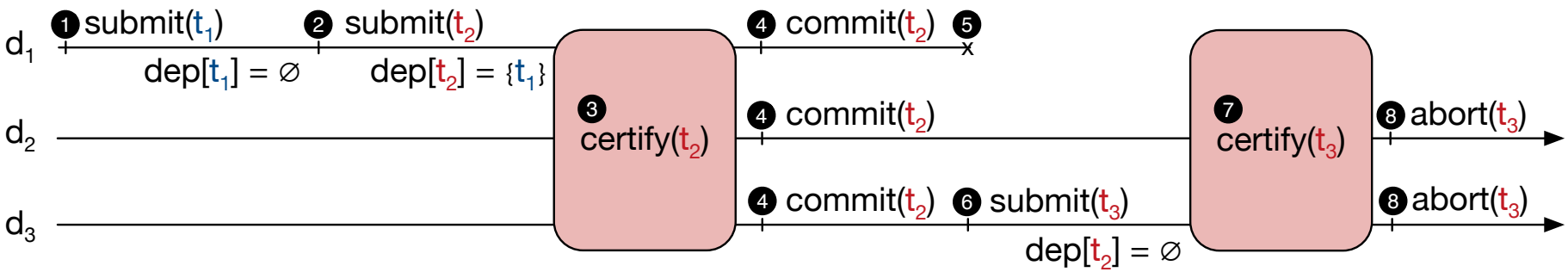
before t_1 is replicated to correct data center d_3.



Liveness of Strong Transactions

Data center d_1 crashes

before t_1 is replicated to correct data center d_3.

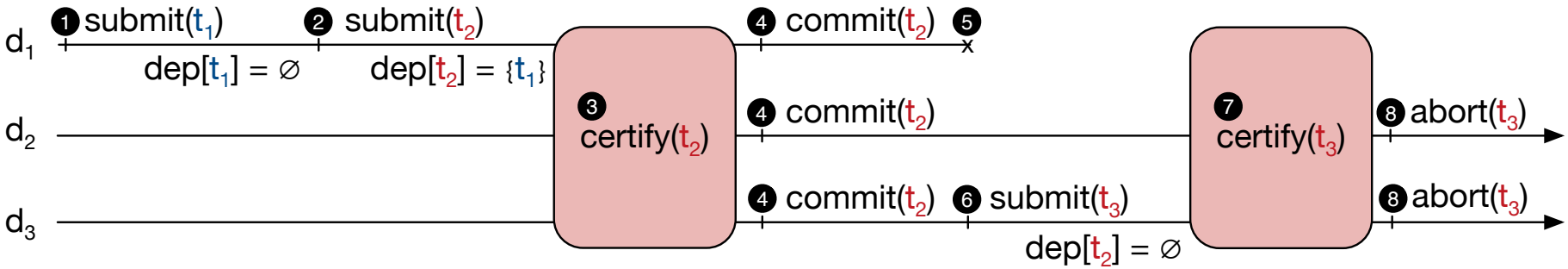


Transaction t_2 will never be visible at d_3.

No transaction t_3 conflicting with t_2 can commit.

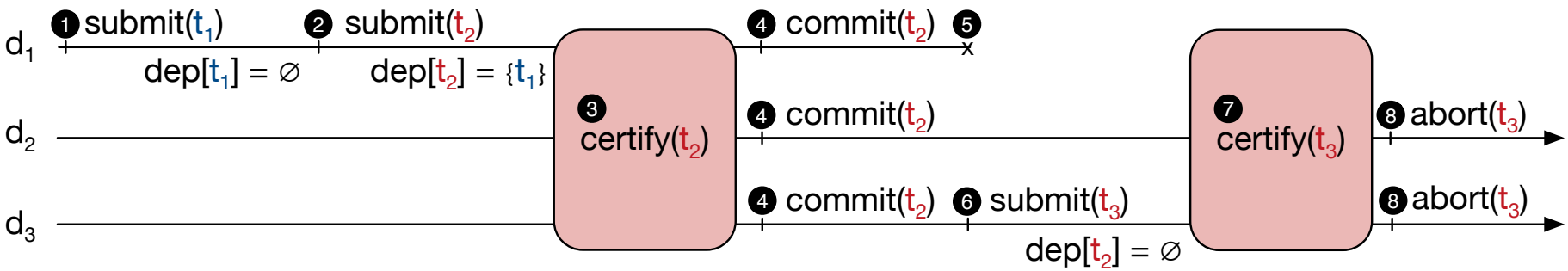
Liveness of Strong Transactions

UniStore ensures that before a strong transaction commits,
all its causal dependencies are uniform,
i.e., will eventually become visible at all correct data centers.



Liveness of Strong Transactions

UniStore ensures that before a strong transaction commits,
all its causal dependencies are uniform,
i.e., will eventually become visible at all correct data centers.



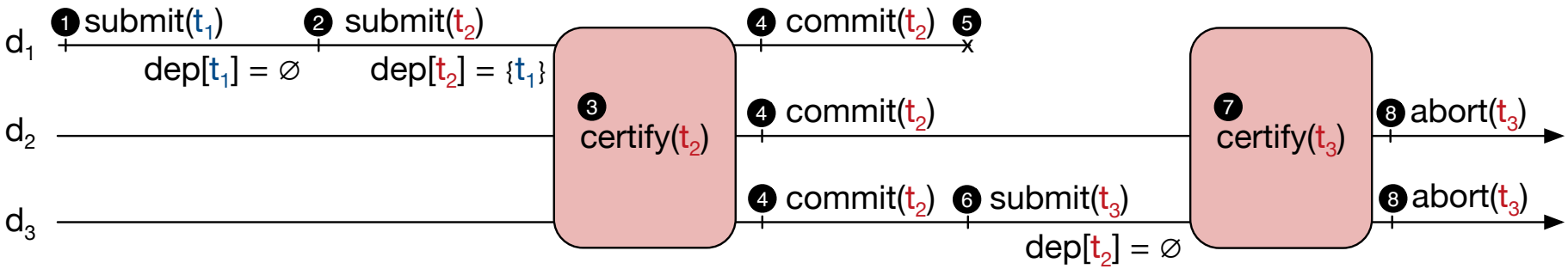
Transaction t_1 will eventually be visible at d_3.

Transaction t_2 will eventually be visible at d_3.

Transaction t_3 may be committed at d_3.

Performance of UniStore

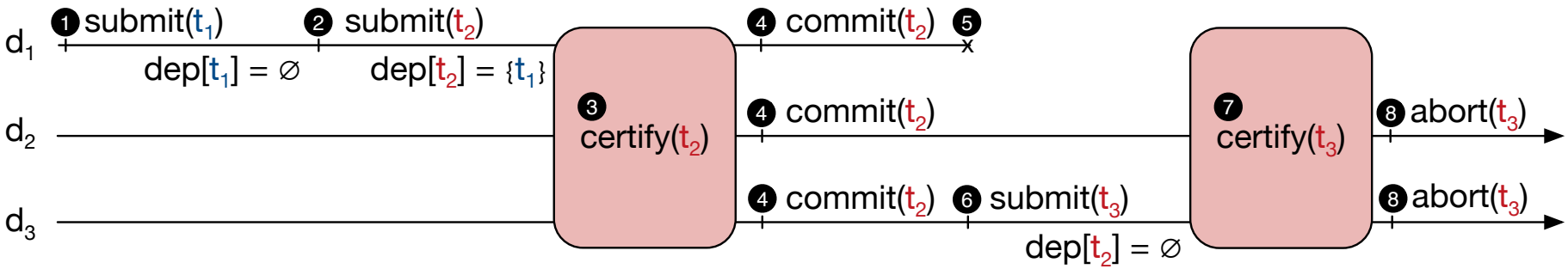
Causal transactions remain highly-available, i.e., committed locally.



A strong transaction may have to wait for some of its dependencies to become uniform before committing.

Performance of UniStore

Causal transactions remain highly-available, i.e., committed locally.



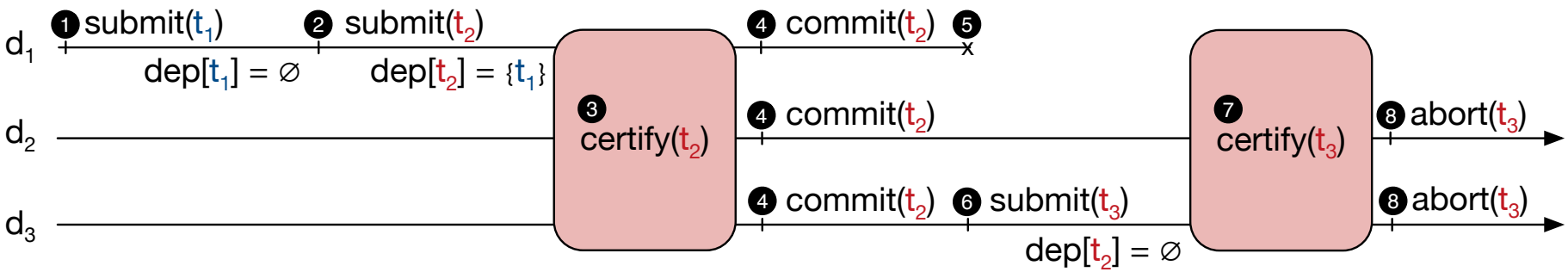
A strong transaction may have to wait for some of its dependencies to become uniform before committing.

However, this may cost too much.

Performance of UniStore

UniStore makes a remote causal transaction visible to clients only after it is uniform.

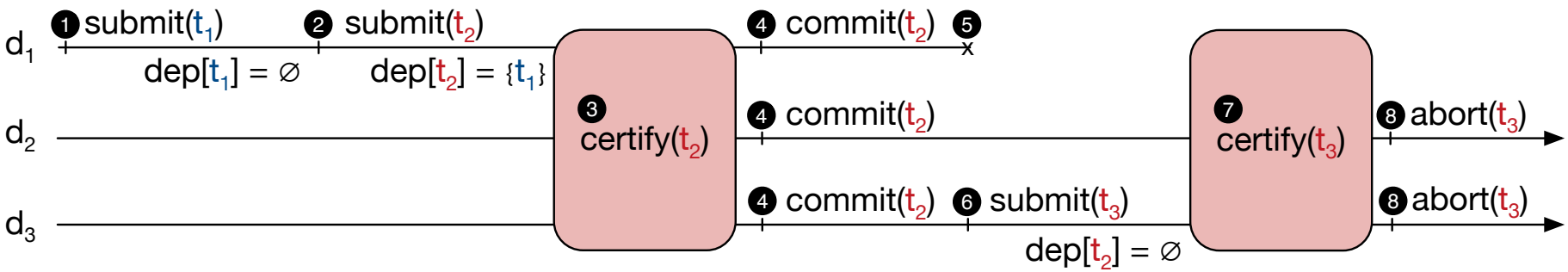
Causal transactions are executed on an (almost) uniform snapshot that may be slightly in the past.



Performance of UniStore

UniStore makes a remote causal transaction visible to clients only after it is uniform.

Causal transactions are executed on an (almost) uniform snapshot that may be slightly in the past.



A strong transaction only needs to wait for causal transactions originating at the local data center to become uniform.

Scalability of UniStore

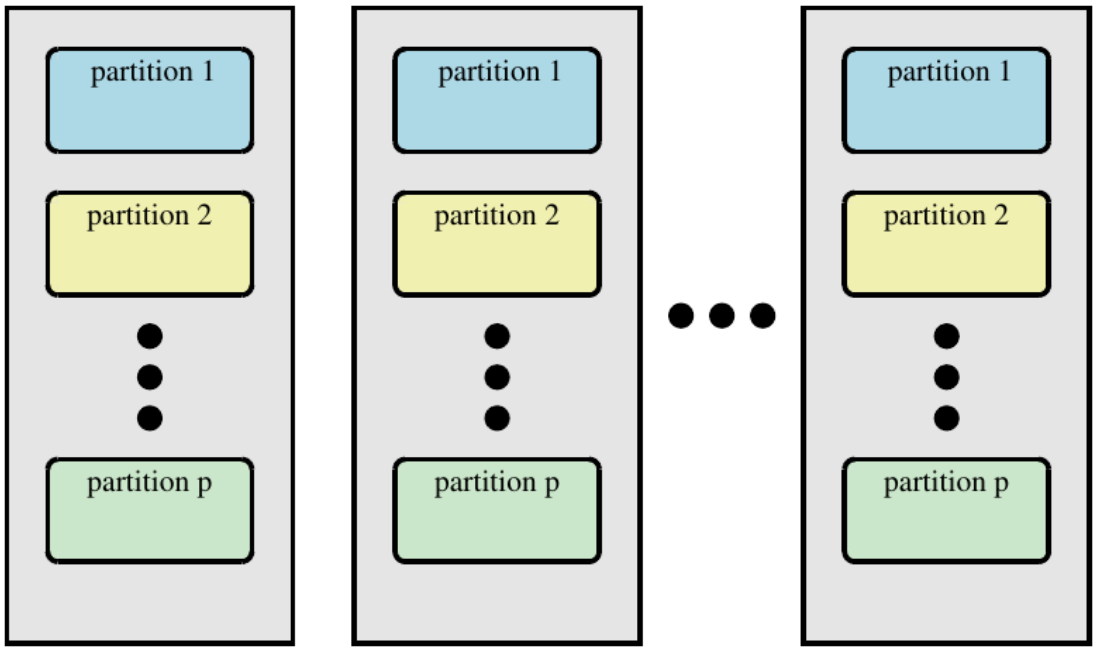
UniStore scales horizontally,

i.e., with the number of machines (partitions) in each data center.

System Model

$\mathcal{D} = \{1, \ldots, D\}$: the set of data centers

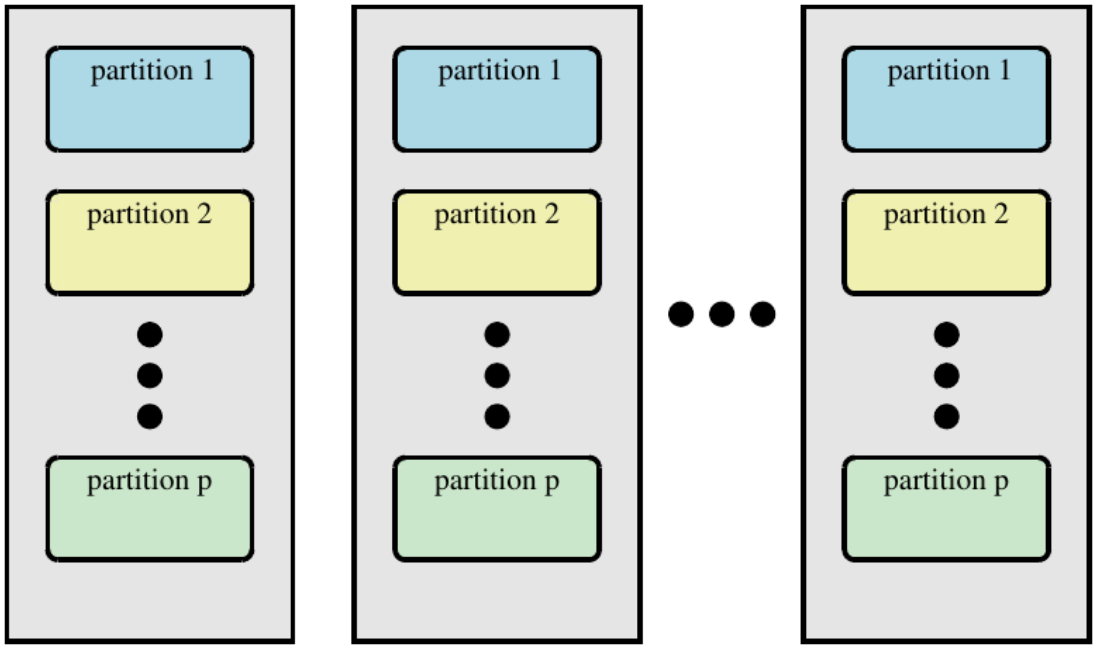
$\mathcal{P} = \{1, \ldots, N\}$: the set of (logical) partitions



p_d^m : the replica of partition m at data center d

System Model

$D = 2f + 1$ and $\leq f$ data centers may fail



Any two replicas are connected by a reliable FIFO channel.

Messages between correct data centers will eventually be delivered.

System Model

Replicas have loosely synchronized physical clocks.

The correctness of UniStore does *not*
depend on the precision of clock synchronization.

Fault-tolerant Causal Consistency Protocol

Requirement: Tracking Uniformity

UniStore makes a remote causal transaction visible to clients only after it is uniform.

Requirement: Tracking Uniformity

UniStore makes a remote causal transaction visible to clients only after it is uniform.

Definition (Uniform)

A transaction is uniform if both the transaction and its causal dependencies are guaranteed to be eventually replicated at all correct data centers.

Requirement: Tracking Uniformity

UniStore makes a remote causal transaction visible to clients only after it is uniform.

Definition (Uniform)

A transaction is uniform if both the transaction and its causal dependencies are guaranteed to be eventually replicated at all correct data centers.

A transaction is considered uniform once it is visible at $f+1$ data centers.

Metadata for Causal Transactions

Each transaction is tagged with a commit vector $commitVec$.

$$commitVec \in [\mathcal{D} \to \mathbb{N}]$$

For a transaction originating at data center d,
we call $commitVec[d]$ its local timestamp.

Metadata for Causal Transactions

Each transaction is tagged with a commit vector $commitVec$.

$$commitVec \in [\mathcal{D} \to \mathbb{N}]$$

For a transaction originating at data center d,
we call $commitVec[d]$ its local timestamp.

Commit vectors are sent to sibling replicas
via replication and forwarding.

Metadata for Causal Transactions

Each replica p_d^m maintains the following three vectors:

$$\mathsf{knownVec} \in [\mathcal{D} \to \mathbb{N}]$$

$$\mathsf{stableVec} \in [\mathcal{D} \to \mathbb{N}]$$

$$\mathsf{uniformVec} \in [\mathcal{D} \to \mathbb{N}]$$

Metadata for Causal Transactions

$$\mathsf{knownVec} \in [\mathcal{D} \to \mathbb{N}]$$

Property (Property of knownVec)

For each data center i,

the replica p_d^m stores the updates to partition m

by transactions originating at i with local timestamps $\leq \mathsf{knownVec}[i]$.

Metadata for Causal Transactions

$$\mathsf{stableVec} \in [\mathcal{D} \to \mathbb{N}]$$

Property (Property of stableVec)

For each data center i,

the data center d stores the updates

by transactions originating at i with local timestamps $\leq \mathsf{stableVec}[i]$.

Metadata for Causal Transactions

$$\mathsf{uniformVec} \in [\mathcal{D} \to \mathbb{N}]$$

Property (Property of uniformVec)

All update transactions originating at i
with local timestamps $\leq \mathsf{uniformVec}[i]$
are replicated at $f+1$ data centers including d.

Metadata for Causal Transactions

$$\mathsf{uniformVec} \in [\mathcal{D} \to \mathbb{N}]$$

Lemma

All update transactions
with commit vectors $\leq$ uniformVec *are uniform.*

Metadata for Causal Transactions

$$\mathsf{uniformVec} \in [\mathcal{D} \to \mathbb{N}]$$

Lemma

All update transactions
with commit vectors $\leq$ uniformVec *are uniform.*

UniStore makes a remote causal transaction visible to clients only after it is uniform.

Causal Consistency Protocol: Start

Causal transactions are executed on an (almost) uniform snapshot.

$\mathsf{snapVec}[tid][d]$ ensures "read-your-writes".

Causal Consistency Protocol: Read

Causal transactions are executed on an (almost) uniform snapshot.

wait : ensure that it is as up-to-date as required by the snapshot

Causal Consistency Protocol: Commit

Read-only transactions returns immediately.

2PC protocol for update transactions

Adding Strong Transactions

Requirement: ConflictOrdering

$$\forall t_1, t_2 \in T_{strong}.\ t_1 \bowtie t_2 \implies t_1 \prec t_2 \vee t_2 \prec t_1.$$

Each strong transaction is assigned a scalar strong timestamp.

$$commitVec \in [\mathcal{D} \cup \{strong\} \to \mathbb{N}]$$

Metadata for Strong Transactions

$$\mathsf{knownVec} \in [\mathcal{D} \cup \{strong\} \to \mathbb{N}]$$

Property (Property of $\mathsf{knownVec}[strong]$)

Replica p_d^m stores the updates to m by all strong transactions with $commitVec[strong] \leq \mathsf{knownVec}[strong]$.

Metadata for Strong Transactions

$$\mathsf{stableVec} \in [\mathcal{D} \cup \{strong\} \to \mathbb{N}]$$

5: **when received** KNOWNVEC_LOCAL($l, knownVec$)
6: $\quad \mathsf{localMatrix}[l] \leftarrow knownVec$
7: $\quad$ **for** $i \in \mathscr{D}$ **do**
8: $\quad\quad \mathsf{stableVec}[i] \leftarrow \min\{\mathsf{localMatrix}[n][i] \mid n \in \mathscr{P}\}$
9: $\quad \mathsf{stableVec}[strong] \leftarrow \min\{\mathsf{localMatrix}[n][strong] \mid n \in \mathscr{P}\}$

Property (Property of $\mathsf{stableVec}[strong]$)

Data center d stores the updates by all strong transactions with $commitVec[strong] \leq \mathsf{knownVec}[strong]$.

Metadata for Strong Transactions

$$\mathsf{uniformVec} \in [\mathcal{D} \to \mathbb{N}]$$

10: **when received** STABLEVEC($i, stableVec$)
11: stableMatrix$[i] \leftarrow stableVec$
12: $G \leftarrow$ all groups with $f+1$ replicas that include p_d^m
13: **for** $j \in \mathscr{D}$ **do**
14: **var** $ts \leftarrow \max\{\min\{\mathsf{stableMatrix}[h][j] \mid h \in g\} \mid g \in G\}$
15: uniformVec$[j] \leftarrow \max\{\mathsf{uniformVec}[j], ts\}$

The commit protocol for strong transactions guarantees their uniformity.

Strong Consistency Protocol: Commit

```
1: function COMMIT_STRONG(tid, c)
2:     UNIFORM_BARRIER(snapVec[tid])
3:     ⟨d, vc, c⟩ ← CERTIFY(tid, wbuff[tid], rset[tid], snapVec[tid], c)
4:     lc ← max{lc, c} + 1
5:     return ⟨d, vc, lc⟩
```

A strong transaction only needs to wait for causal transactions originating at the local data center to become uniform.

```
20: function UNIFORM_BARRIER(V, c)
21:     lc ← max{lc, c} + 1
22:     wait until uniformVec[d] ≥ V[d]
23:     return lc
```

Strong Consistency Protocol: Commit

```
1: function COMMIT_STRONG(tid, c)
2:     UNIFORM_BARRIER(snapVec[tid])
3:     ⟨d, vc, c⟩ ← CERTIFY(tid, wbuff[tid], rset[tid], snapVec[tid], c)
4:     lc ← max{lc, c} + 1
5:     return ⟨d, vc, lc⟩
```

$$\langle d \in \{\text{COMMIT}, \text{ABORT}\}, vc \rangle \leftarrow \text{CERTIFY}(t)$$

Strong Consistency Protocol: Commit

```
1: function COMMIT_STRONG(tid, c)
2:     UNIFORM_BARRIER(snapVec[tid])
3:     ⟨d, vc, c⟩ ← CERTIFY(tid, wbuff[tid], rset[tid], snapVec[tid], c)
4:     lc ← max{lc, c} + 1
5:     return ⟨d, vc, lc⟩
```

$$\langle d \in \{\text{COMMIT}, \text{ABORT}\}, vc \rangle \leftarrow \text{CERTIFY}(t)$$

Multi-Shot Distributed Transaction Commit

Gregory Chockler
Royal Holloway, University of London, UK

Alexey Gotsman[1]
IMDEA Software Institute, Madrid, Spain

White-Box Atomic Multicast

Alexey Gotsman
IMDEA Software Institute

Anatole Lefort
Télécom SudParis

Gregory Chockler
Royal Holloway, University of London

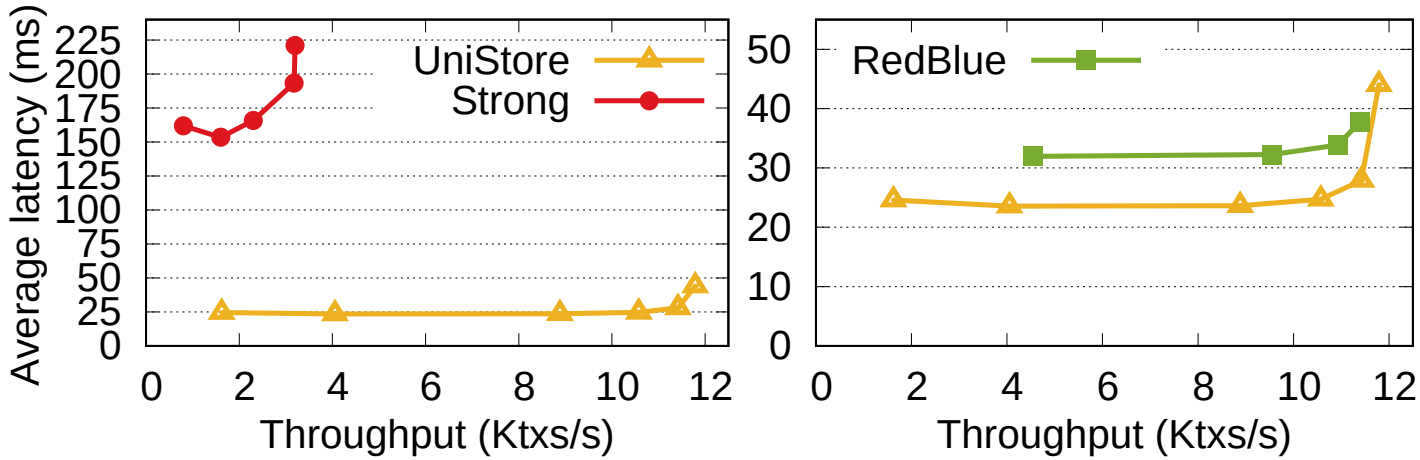
2PC across partitions + Paxos among replicas of each partition

uses white-box optimizations that minimize the commit latency

Evaluation

Performance of UniStore

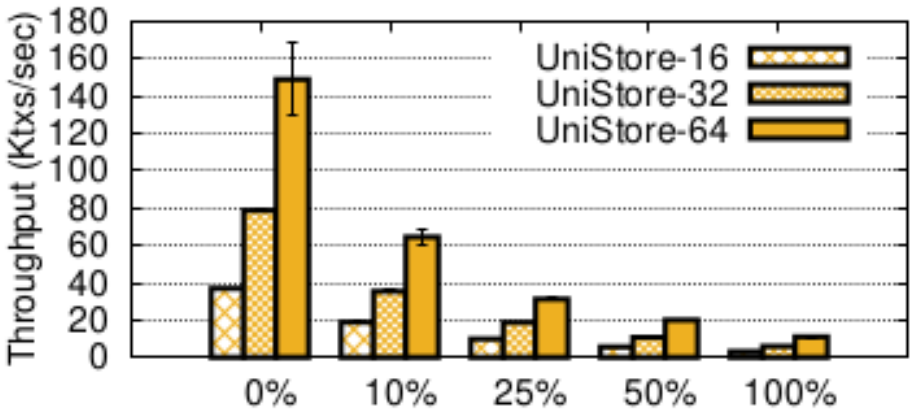
Throughput: 5% and 259% higher than RedBlue and Strong



RUBiS benchmark: throughput vs. average latency.

Latency: 24ms vs. 32ms of RedBlue and 162ms of Strong

Scalability of UniStore



Scalability when varying the ratio of strong transactions.

UniStore is able to scales almost linearly.

Evaluation

For more evaluations, please refer to the paper.

Conclusion

UniStore is a fast, scalable, and fault-tolerant

transactional distributed key-value store

that supports a combination of weak and strong consistency.

Conclusion

UniStore is a fast, scalable, and fault-tolerant

transactional distributed key-value store

that supports a combination of weak and strong consistency.

"We expect the key ideas in UniStore to pave the way
for practical systems that combine causal and strong consistency."

Hengfeng Wei (hfwei@nju.edu.cn)