

# Consistency in Transactional Distributed Databases: Protocols and Testing

Hengfeng Wei (魏恒峰)

hfwei@nju.edu.cn

Nov. 04, 2022



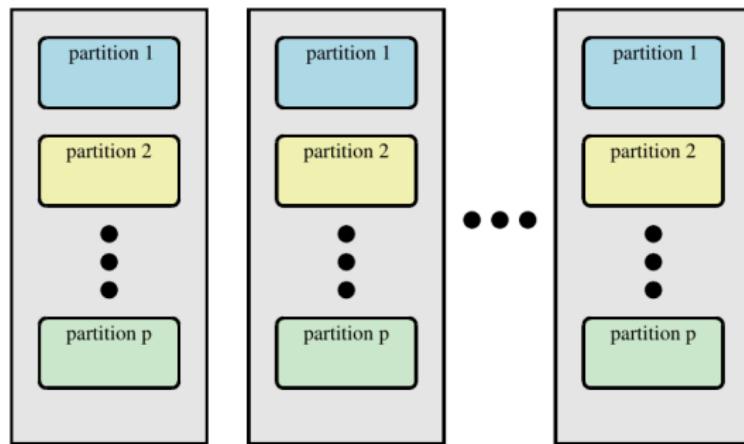
# Background

Centralized Databases *vs.* Distributed Databases



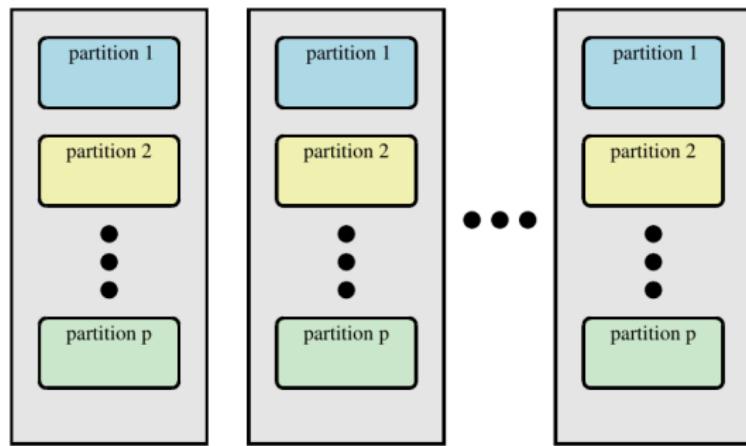
# Data Consistency Problem

The “Data Partition + Data Replication” Architecture



# Data Consistency Problem

The “Data Partition + Data Replication” Architecture



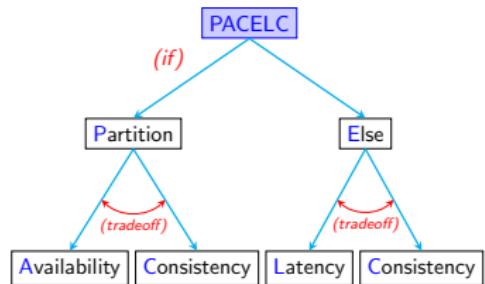
(Distributed) Data Consistency Problem

# Data Consistency Problem

(Strong) Consistency, Availability, Latency、Partition tolerance



CAP Theorem  
(Brewer@PODC2000)



PACELC Tradeoff  
(Abadi@Computer2012)

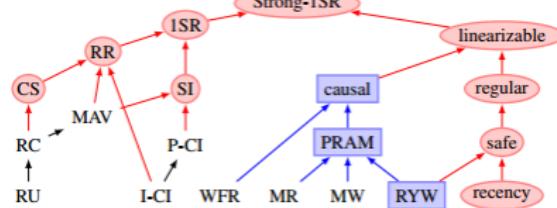
# Data Consistency Problem



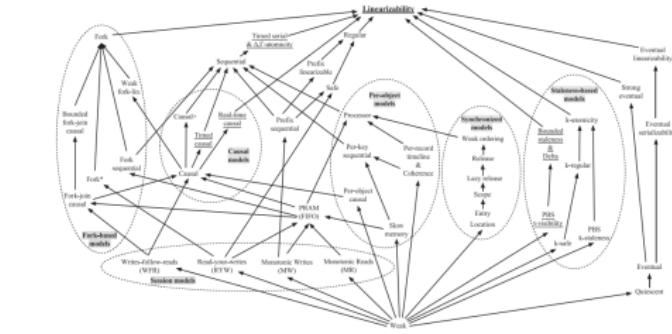
没有一劳永逸的解决方案

# Consistency Models

Use various **consistency models** to capture these tradeoffs



(Bailis@VLDB2012)



(Viotti@CSUR2016)

**STRONG**  
**WEAK**

# My Researches

On the data consistency theory 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 algorithms?

# My Researches

On the data consistency theory 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 algorithms?

Classic problems with the ever-changing requirements

# My Researches (I)

Read/Write Register<sup>a</sup> ( $\geq 2014$ )



Distributed Non-transactional Key-Value Stores

---

<sup>a</sup> 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 (I)

Wei:TPDS2016

Wei:TC2017

Wei:PODC2020-Huang

# My Researches (II)

Replicated Data Types<sup>b</sup> ( $\geq 2017$ )



(a) Google Docs



(b) Apache Wave



(c) Wikipedia



(d) LaTeX Editor

<sup>b</sup> 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<sup>c</sup> ( $\geq 2020$ )



<sup>c</sup>分布式事务。每个事务由一组操作构成，这些操作要么都成功，要么都失败。 ↗ ↘ ↙

# 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

*IMDEA Software Institute*

Borja de Régil

Hengfeng Wei \*

*Nanjing University*

ATC'2021 (CCF A)

UNISTORE is the first **fault-tolerant** and scalable **transactional** data store that **combines** weak and strong consistency.

# Overview of the Work on UNISTORE

Partial Order-Restrictions (PoR) Consistency

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

CC: CAUSALCONSISTENCY; SER: SERIALIZABILITY

# Overview of the Work on UNISTORE

Partial Order-Restrictions (PoR) Consistency

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

CC: CAUSALCONSISTENCY; SER: SERIALIZABILITY

Challenges (I): To ensure **liveness** of the system in the presence of data center failures

Challenges (II): To provide a rigorous correctness **proof** of the protocol which is rather complicated

# Overview of the Work on UNISTORE

## UNISTORE: A fault-tolerant marriage of causal and strong consistency

Manuel Bravo

Alexey Gotsman

Borja de Régil

*IMDEA Software Institute*

Hengfeng Wei \*

*Nanjing University*

ATC'2021 (CCF A)

I am fully responsible for developing a 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<sup>d</sup>, and proposed solutions to fix them

---

<sup>d</sup>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.

# 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 among data centers

# The UNI- Approach

- ▶ Multiple consistency levels coexist in a store

# The UNI- Approach

- ▶ Multiple consistency levels coexist in a store
- ▶ Take **weak consistency** as the default and the baseline

# The UNI- Approach

- ▶ Multiple consistency levels coexist in a store
- ▶ Take **weak consistency** as the default and the baseline
- ▶ Programmers can choose the transactions that should be executed under **strong consistency**

# The UNI- Approach

- ▶ Multiple consistency levels coexist in a store
- ▶ Take **weak consistency** as the default and the baseline
- ▶ Programmers can choose the transactions that should be executed under **strong consistency**
  - ▶ e.g., if the execution of a set of transactions may violate the application violations

# The UNI- Approach

Partial Order-Restrictions (PoR) Consistency  
(Li@OSDI'2012, Li@ACT'2018)

# The UNI- Approach

Partial Order-Restrictions (PoR) Consistency  
(Li@OSDI'2012, Li@ACT'2018)

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

# The UNI- Approach

## Partial Order-Restrictions (PoR) Consistency (Li@OSDI'2012, Li@ACT'2018)

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

# The UNI- Approach

## Partial Order-Restrictions (PoR) Consistency (Li@OSDI'2012, Li@ACT'2018)

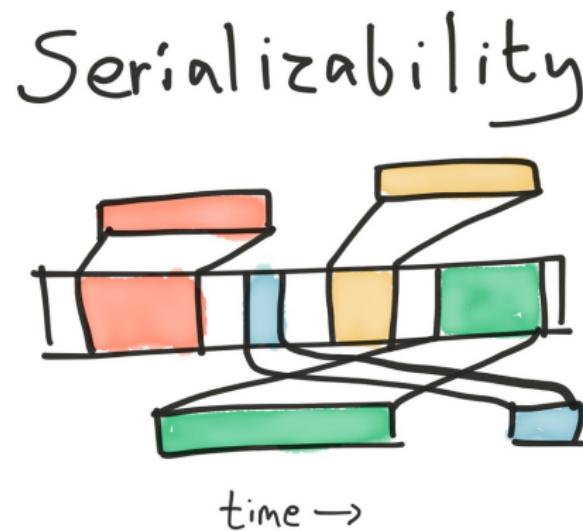
- ▶ PoR allows programmers to classify transactions as either causal or strong.
- ▶ Causal transactions satisfy CAUSALCONSISTENCY:
  - ▶ Clients see updates in an order that respects the potential causality between them.
  - ▶ Causally independent transactions can be executed concurrently.
- ▶ Programmers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

# CAUSAL CONSISTENCY

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

Causally independent transactions can be executed concurrently.

# SERIALIZABILITY



All transactions seem to be executed in some **sequential** order.

# A Banking Application

DEPOSIT

WITHDRAW

QUERY

INTEREST



Invariant:  $\text{balance} \geq 0$

# A Banking Application

DEPOSIT    WITHDRAW    QUERY    INTEREST



Invariant:  $\text{balance} \geq 0$

DEPOSIT operations can be executed under CAUSALCONSISTENCY.

# A Banking Application

DEPOSIT    WITHDRAW    QUERY    INTEREST



Invariant:  $\text{balance} \geq 0$

DEPOSIT operations can be executed under CAUSALCONSISTENCY.

DEPOSIT(50)

# A Banking Application

DEPOSIT

WITHDRAW

QUERY

INTEREST



Invariant:  $\text{balance} \geq 0$

# A Banking Application

DEPOSIT    WITHDRAW    QUERY    INTEREST



Invariant:  $\text{balance} \geq 0$

However, CAUSALCONSISTENCY also allows two causally independent **WITHDRAW** to execute concurrently, without knowing each other.

# A Banking Application

DEPOSIT    WITHDRAW    QUERY    INTEREST



Invariant:  $\text{balance} \geq 0$

However, CAUSALCONSISTENCY 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 transactions.
- ▶ Any transaction involved in the conflict relation is marked **strong**.
- ▶ PoR ensures that conflicting transactions are executed serially.

# A Banking Application

DEPOSIT    WITHDRAW    QUERY    INTEREST



Invariant:  $\text{balance} \geq 0$

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)

## EVENTUAL VISIBILITY

A transaction that is either **strong** or **causal** that originates at a **correct data center** eventually becomes **visible** at all **correct** data centers.

# Design Challenges of UNISTORE

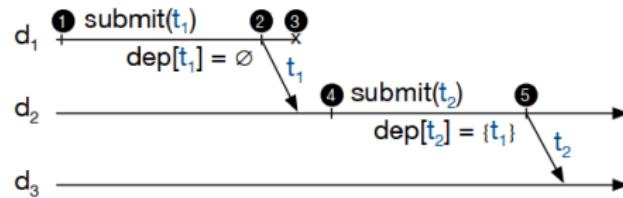
How to ensure **liveness** despite data center failures?



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

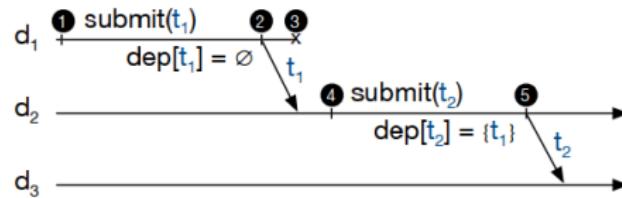
# Liveness of Causal Transactions

$d_1$  crashes before  $t_1$  is replicated to  $d_3$ .



# Liveness of Causal Transactions

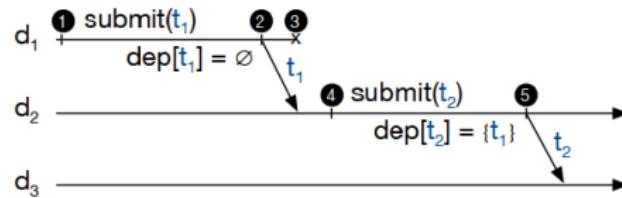
$d_1$  crashes before  $t_1$  is replicated to  $d_3$ .



Transaction  $t_2$  may never become visible at  $d_3$ .

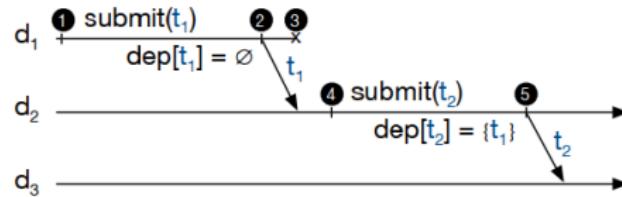
# Liveness of Causal Transactions

$d_1$  crashes before  $t_1$  is replicated to  $d_3$ .



# Liveness of Causal Transactions

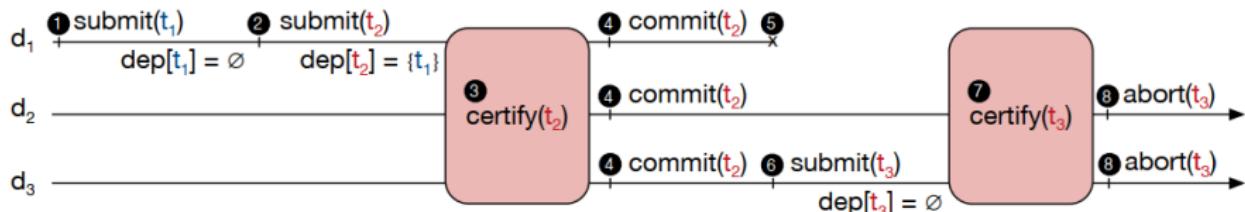
$d_1$  crashes before  $t_1$  is replicated to  $d_3$ .



$d_2$  need to **forward** causal transactions to other data centers.

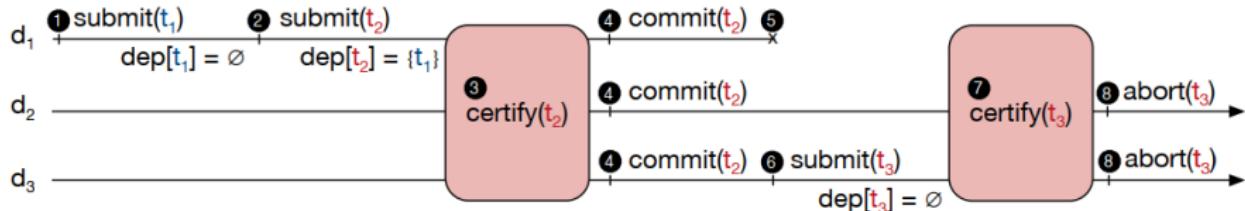
# Liveness of Strong Transactions

$d_1$  crashes before  $t_1$  is replicated to  $d_3$ .



# Liveness of Strong Transactions

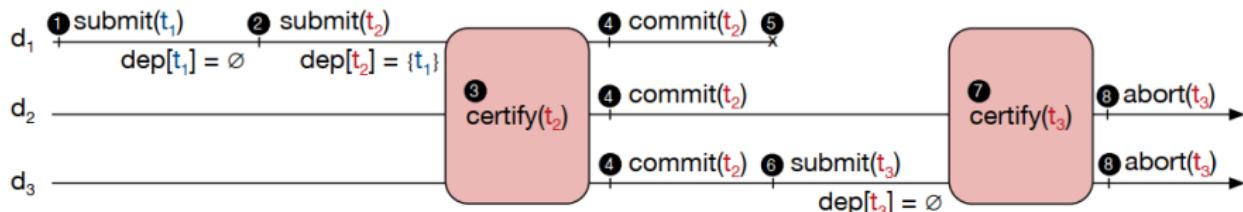
$d_1$  crashes before  $t_1$  is replicated to  $d_3$ .



$t_2$  will never be visible at  $d_3$ .

# Liveness of Strong Transactions

$d_1$  crashes before  $t_1$  is replicated to  $d_3$ .

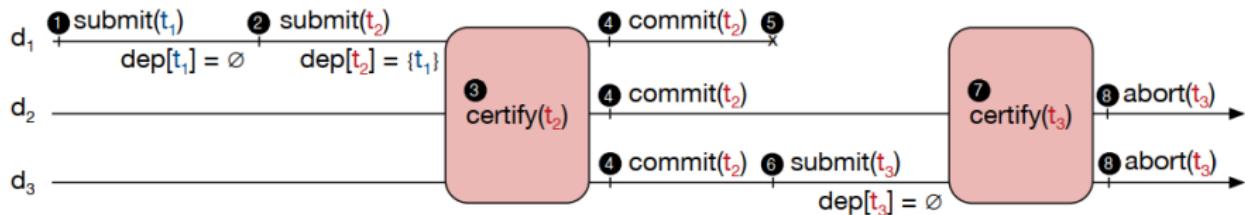


$t_2$  will never be visible at  $d_3$ .

No transaction  $t_3$  conflicting with  $t_2$  can commit.

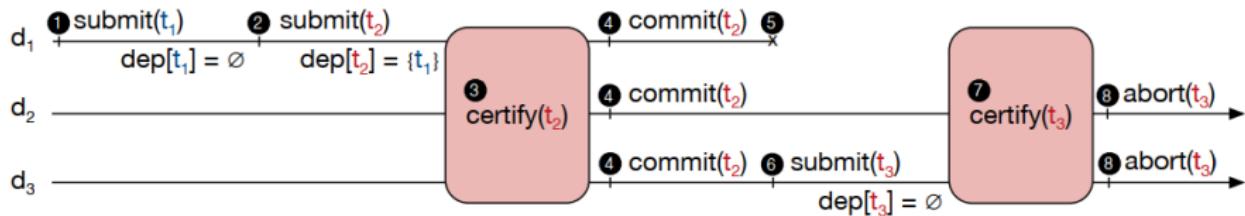
# Liveness of Strong Transactions

A **strong** transaction should wait for all its **causal dependencies** to become **uniform** before committing.  
(eventually become visible at all correct data centers)



# Liveness of Strong Transactions

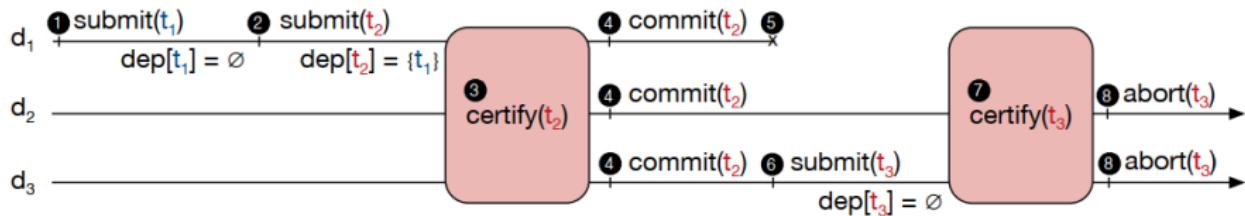
A **strong** transaction should wait for all its **causal dependencies** to become **uniform** before committing.  
(eventually become visible at all correct data centers)



$t_1$  will eventually be visible at  $d_3$ .

# Liveness of Strong Transactions

A **strong** transaction should wait for all its **causal dependencies** to become **uniform** before committing.  
(eventually become visible at all correct data centers)

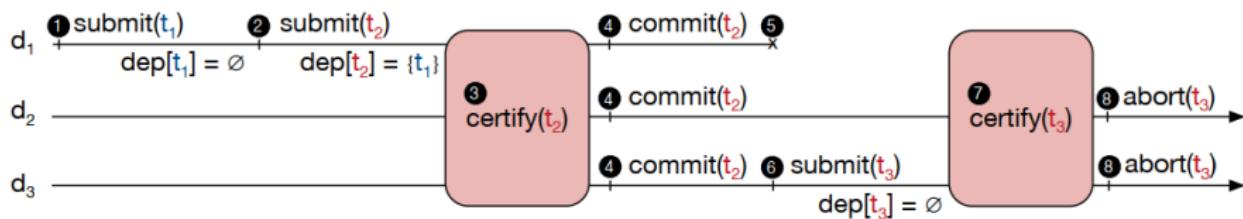


$t_1$  will eventually be visible at  $d_3$ .

$t_2$  will eventually be visible at  $d_3$ .

# Liveness of Strong Transactions

A **strong** transaction should wait for all its **causal dependencies** to become **uniform** before committing.  
(eventually become visible at all correct data centers)



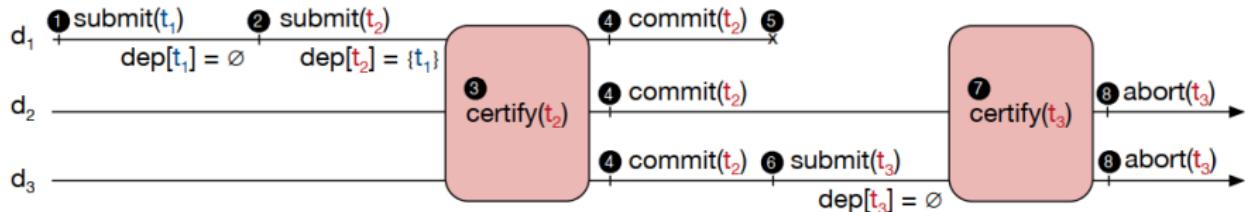
$t_1$  will eventually be visible at  $d_3$ .

$t_2$  will eventually be visible at  $d_3$ .

$t_3$  may be committed at  $d_3$ .

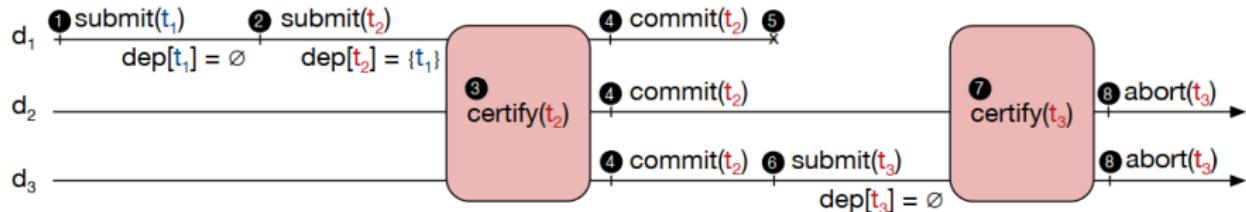
# Minimizing the Latency of Strong Transactions

A **strong** transaction should wait for all its **causal dependencies** to become **uniform** before committing.



# Minimizing the Latency of Strong Transactions

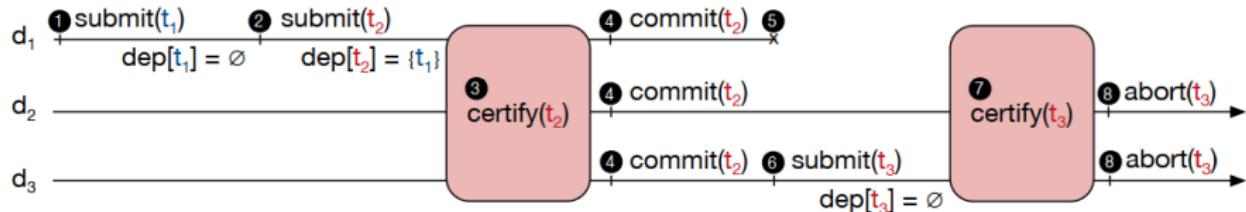
A **strong** transaction should wait for all its **causal dependencies** to become **uniform** before committing.



$t_2$  waits for local  $t_1$  to become uniform before committing.

# Minimizing the Latency of Strong Transactions

A **strong** transaction should wait for all its **causal dependencies** to become **uniform** before committing.

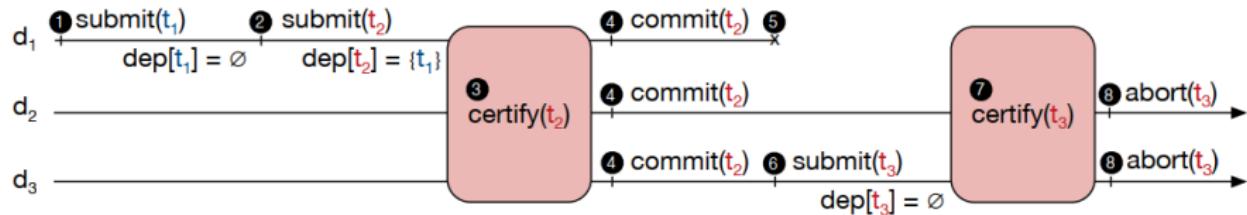


$t_2$  waits for **local**  $t_1$  to become uniform before committing.

However, it may cost too much to wait for **remote causal dependencies** to become uniform.

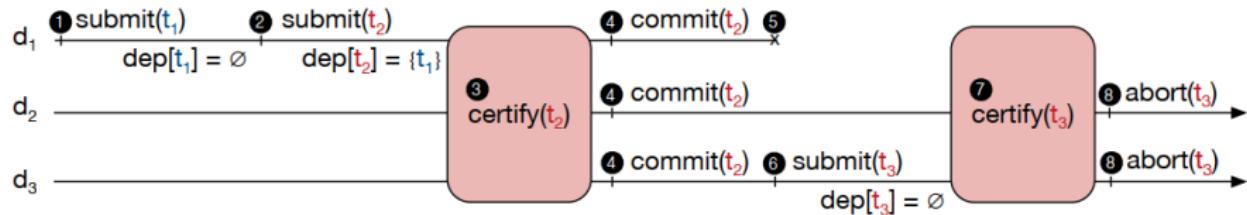
# Minimizing the Latency of Strong Transactions

Let **causal transactions** be executed on an (almost) **uniform snapshot** that may be slightly in the past.



# Minimizing the Latency of Strong Transactions

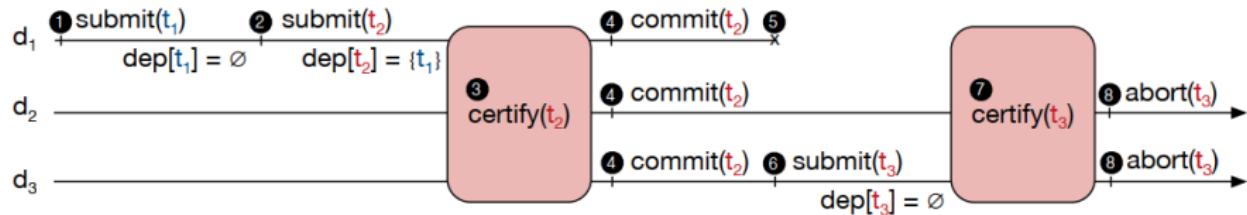
Let **causal transactions** be executed on an (almost) **uniform snapshot** that may be slightly in the past.



Make a **remote causal transaction** visible only **after it is uniform**.

# Minimizing the Latency of Strong Transactions

Let **causal transactions** be executed on an (almost) **uniform snapshot** that may be slightly in the past.



Make a **remote causal transaction** visible only **after it is uniform**.

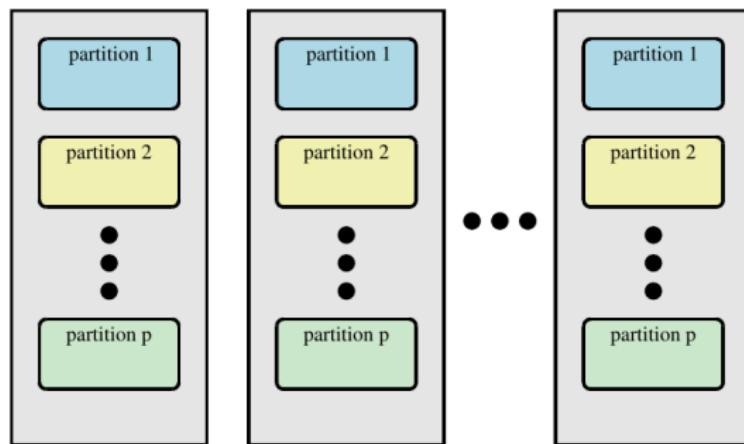
A **strong** transaction does *not* need to wait for **remote causal dependencies** to become uniform before committing.

# Related Work

# System Model and Notations

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

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



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

# Fault-tolerant Causal Consistency Protocol

UNISTORE relies on various vector clocks  $\in [\mathcal{D} \rightarrow \mathbb{N}]$  for

- ▶ Tracking causality (*commitVec*)
- ▶ Representing causally consistent snapshots (*snapshotVec*)
- ▶ Tracking what is replicated where

# Tracking What is Replicated Where

Make a **remote causal transaction** visible only **after it is uniform**.

# Tracking What is Replicated Where

Make a **remote causal transaction** visible 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.

# Tracking What is Replicated Where

Make a **remote causal transaction** visible 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.

# Tracking What is Replicated Where

Each replica  $p_d^m$  maintains three vectors:

**knownVec:** track the set of transactions replicated at  $p_d^m$

# Tracking What is Replicated Where

Each replica  $p_d^m$  maintains three vectors:

**knownVec**: track the set of transactions replicated at  $p_d^m$

**stableVec**: track the set of transactions replicated at the local data center  $d$

# Tracking What is Replicated Where

Each replica  $p_d^m$  maintains three vectors:

**knownVec**: track the set of transactions replicated at  $p_d^m$

**stableVec**: track the set of transactions replicated at the local data center  $d$

**uniformVec**: track the set of transactions replicated at  $f + 1$  data centers

# Fault-tolerant Causal Consistency Protocol

- ▶ Causal transactions are executed on a uniform snapshot.
- ▶ Read-only transactions returns immediately.
- ▶ Update transactions commit using a variant of 2PC protocol.

# Adding Strong Transactions

PoR ensures that conflicting transactions are executed serially.

# Adding Strong Transactions

PoR ensures that conflicting transactions are executed serially.

Assigning to each **strong transaction** a scalar *strong timestamp*.

# Adding Strong Transactions

PoR ensures that conflicting transactions are executed serially.

Assigning to each **strong transaction** a scalar *strong timestamp*.

$$[\mathcal{D} \rightarrow \mathbb{N}] \quad vs. \quad [\mathcal{D} \cup \{\text{strong}\} \rightarrow \mathbb{N}]$$

# Adding Strong Transactions

PoR ensures that conflicting transactions are executed serially.

Assigning to each **strong transaction** a scalar *strong timestamp*.

$$[\mathcal{D} \rightarrow \mathbb{N}] \quad vs. \quad [\mathcal{D} \cup \{\text{strong}\} \rightarrow \mathbb{N}]$$

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

# Adding Strong Transactions

Each replica  $p_d^m$  maintains three vectors:

`knownVec[strong]`: track the set of **strong** transactions replicated at  $p_d^m$

# Adding Strong Transactions

Each replica  $p_d^m$  maintains three vectors:

`knownVec[strong]`: track the set of **strong** transactions replicated at  $p_d^m$

`stableVec[strong]`: track the set of **strong** transactions replicated at the local data center  $d$

# Adding Strong Transactions

Each replica  $p_d^m$  maintains three vectors:

`knownVec[strong]`: track the set of **strong** transactions replicated at  $p_d^m$

`stableVec[strong]`: track the set of **strong** transactions replicated at the local data center  $d$

# Adding Strong Transactions

Each replica  $p_d^m$  maintains three vectors:

`knownVec[strong]`: track the set of **strong** transactions replicated at  $p_d^m$

`stableVec[strong]`: track the set of **strong** transactions replicated at the local data center  $d$

We do not extend `uniformVec`, since the commit protocol for **strong** transactions automatically guarantees their uniformity.

# Adding Strong Transactions

- ▶ A **strong** transaction waits for local causal dependencies to become uniform before committing.

# Adding Strong Transactions

- ▶ A **strong** transaction waits for local causal dependencies to become uniform before committing.
- ▶ Uses optimistic concurrency control protocol

# Adding Strong Transactions

- ▶ A **strong** transaction waits for local causal dependencies to become uniform before committing.
- ▶ Uses optimistic concurrency control protocol
  - ▶ First *speculatively* execute **strong** transactions locally

# Adding Strong Transactions

- ▶ A **strong** transaction waits for local causal dependencies to become uniform before committing.
- ▶ Uses optimistic concurrency control protocol
  - ▶ First *speculatively* execute **strong** transactions locally
  - ▶ Then certify them globally using a *certification service*

# Strong Consistency Protocol: Commit

## Multi-Shot Distributed Transaction Commit

Gregory Chockler

Royal Holloway, University of London, UK

Alexey Gotsman<sup>1</sup>

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

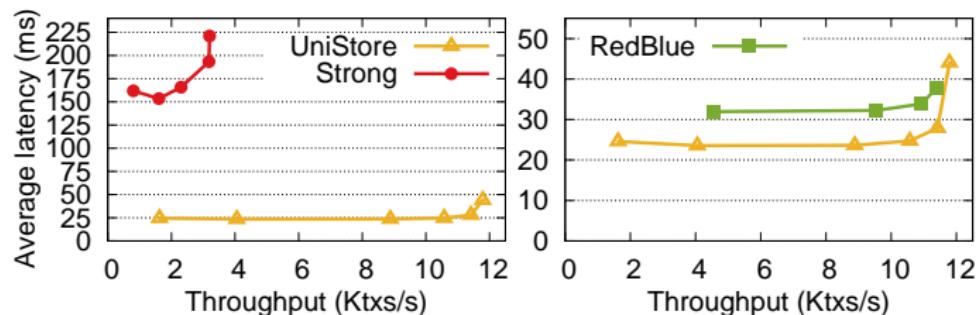
- ▶ 2PC across partitions
- ▶ Paxos among replicas of each partition
- ▶ Uses white-box optimizations that minimize the commit latency

# Correctness Proof

# Evaluation

# Performance of UNISTORE

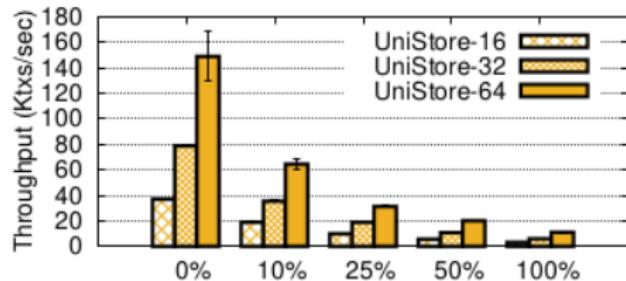
Throughput: 5% and 259% higher than REDBLUE and STRONG



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 scales horizontally and almost linearly,  
i.e., with the number of machines (partitions) in each data center.

# 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)

