

# UNISTORE: A Fault-tolerant Marriage of Causal and Strong Consistency

(submitted to ATC'2021)

Hengfeng Wei

hfwei@nju.edu.cn

January 29, 2021



# 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?

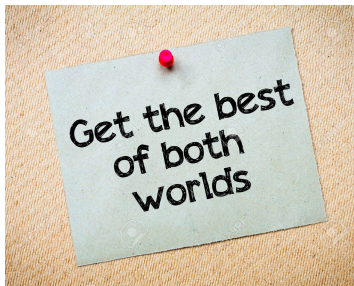
UNiSTORE is a **fast**, **scalable**, and **fault-tolerant**  
**transactional** distributed key-value store  
that supports a **combination of weak and strong consistency**.

Weak consistency: CAUSALCONSISTENCY

Strong consistency: SERIALIZABILITY

# Why UNI-?

Weak consistency: low latency, high availability



Strong consistency: easy to preserve critical application invariants

# Why UNI-?

DEPOSIT

WITHDRAW

QUERY

INTEREST



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

# Why UNI-?

DEPOSIT

WITHDRAW

QUERY

INTEREST



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

Causal consistency allows two concurrent WITHDRAW to execute without knowing each other.

# Why UNI-?

DEPOSIT

WITHDRAW

QUERY

INTEREST



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

Causal consistency allows two concurrent WITHDRAW to execute without knowing each other.

Only WITHDRAW needs to use strong consistency.

# Consistency Model of UNiSTORE

UNiSTORE implements a transactional variant of  
Partial Order-Restrictions (PoR) consistency [Li@ACT'2018]

- (I) transactional causal consistency by default
- (II) to specify conflicting transactions under strong consistency



# Consistency Model of UNiSTORE

## Definition (Session Order)

A transaction  $t_1$  precedes a transaction  $t_2$  in the **session order**, denoted  $t_1 \xrightarrow{so} t_2$ , if they are executed by the same client and  $t_1$  is executed before  $t_2$ .

## Definition (Conflict Relation)

The **conflict relation**, denoted  $\bowtie$ , between transactions is a symmetric relation.

$$t_1 \bowtie t_2 \iff t_2 \bowtie t_1.$$

# Consistency Model of UNISTORE

## Definition (PoR)

A set of transactions  $T \triangleq T_{causal} \uplus T_{strong}$  committed by UNISTORE satisfies **PoR** if there exists a **causal order**  $\prec$  on  $T$  such that

# Consistency Model of UNISTORE

## Definition (PoR)

A set of transactions  $T \triangleq T_{causal} \uplus T_{strong}$  committed by UNISTORE satisfies **PoR** if there exists a **causal order**  $\prec$  on  $T$  such that

**CAUSALITY:** ‘ $\prec$ ’ is a partial order and  $so \subseteq \prec$ .

# Consistency Model of UNISTORE

## Definition (PoR)

A set of transactions  $T \triangleq T_{causal} \uplus T_{strong}$  committed by UNISTORE satisfies **PoR** if there exists a **causal order**  $\prec$  on  $T$  such that

**CAUSALITY:** ' $\prec$ ' is a partial order and  $so \subseteq \prec$ .

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

# Consistency Model of UNISTORE

## Definition (PoR)

A set of transactions  $T \triangleq T_{causal} \uplus T_{strong}$  committed by UNISTORE satisfies **PoR** if there exists a **causal order**  $\prec$  on  $T$  such that

**CAUSALITY:** ‘ $\prec$ ’ is a partial order and  $so \subseteq \prec$ .

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

**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 **in**  $\prec$  all transactions issued at correct data centers.

# Consistency Model of UNISTORE

## Definition (PoR)

A set of transactions  $T \triangleq T_{causal} \uplus T_{strong}$  committed by UNISTORE satisfies **PoR** if there exists a **causal order**  $\prec$  on  $T$  such that

**CAUSALITY:** ‘ $\prec$ ’ is a partial order and  $so \subseteq \prec$ .

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

**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 **in**  $\prec$  all transactions issued at correct data centers.

**RETVAl:**  $\text{INTRETVAl} \wedge \text{EXTRETVAl}$

# Consistency Model of UNISTORE

Consider a read  $r$  from key  $k$  in a transaction  $t$ .

**INTRETVAL** : read from the latest update on  $k$  preceding  $r$  in  $t$

$$\text{RETVAL} = \text{INTRETVAL} \wedge \text{EXTRETVAL}$$

**EXTRETVAL** : read from the last update on  $k$   
of the latest transaction (in an order consistent with  $\prec$ ) preceding  $t$

# Consistency Model of UNISTORE

DEPOSIT

WITHDRAW

QUERY

INTEREST



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

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



# Design Challenge of UNiSTORE

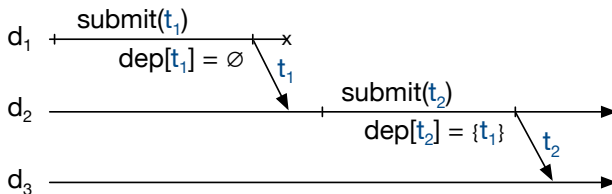
To satisfy **liveness** (**EVENTUAL VISIBILITY**) despite failures



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

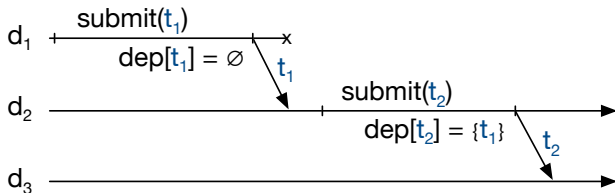
# Design Challenge of UNISTORE (I)

Data center  $d_1$  crashes  
before  $t_1$  is replicated to correct data center  $d_3$ .



# Design Challenge of UNISTORE (I)

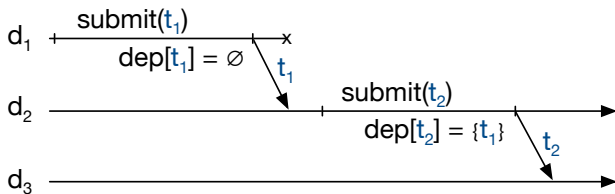
Data center  $d_1$  crashes  
before  $t_1$  is replicated to correct data center  $d_3$ .



Transaction  $t_2$  (at correct data center  $d_2$ )  
may never become visible at correct data center  $d_3$ .

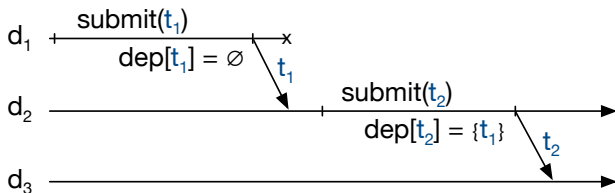
# Fault-tolerance of UNISTORE (I)

Data center  $d_1$  crashes  
before  $t_1$  is replicated to correct data center  $d_3$ .



# Fault-tolerance of UNISTORE (I)

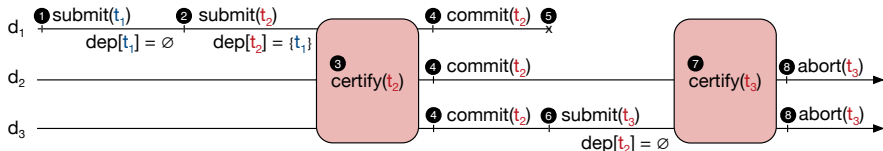
Data center  $d_1$  crashes  
before  $t_1$  is replicated to correct data center  $d_3$ .



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

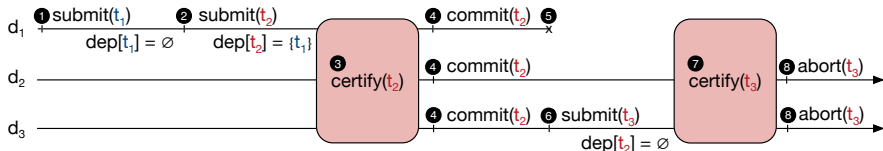
## Design Challenge of UNISTORE (II)

Data center  $d_1$  crashes  
before  $t_1$  is replicated to correct data center  $d_3$ .



## Design Challenge of UNISTORE (II)

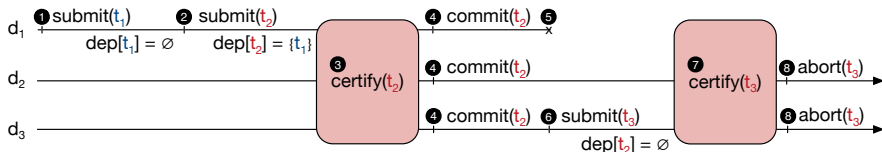
Data center  $d_1$  crashes  
before  $t_1$  is replicated to correct data center  $d_3$ .



Transaction  $t_2$  will never be visible at  $d_3$ .  
No transaction  $t_3$  conflicting with  $t_2$  can commit  
(by CONFLICTORDERING).

## Fault-tolerance of UNISTORE (II)

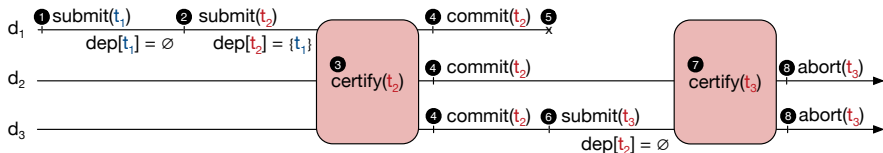
UNISTORE ensures that before a strong transaction commits, all its causal dependencies are **uniform**, i.e., will eventually become visible at all correct data centers.





## Fault-tolerance of UNISTORE (II)

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.



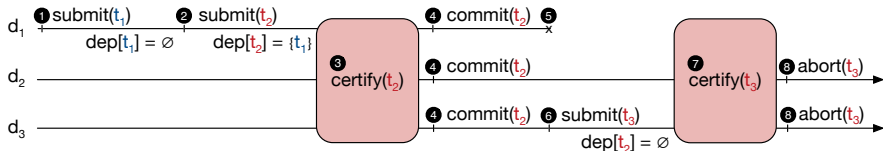
Transaction  $t_1$  will eventually be visible at  $d_3$ .

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

Transaction  $t_3$  may be committed at  $d_3$ .

# Performance of UNISTORE

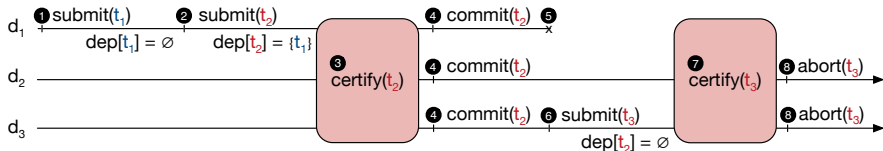
Causal transactions remain highly-available, i.e., committed locally.



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.



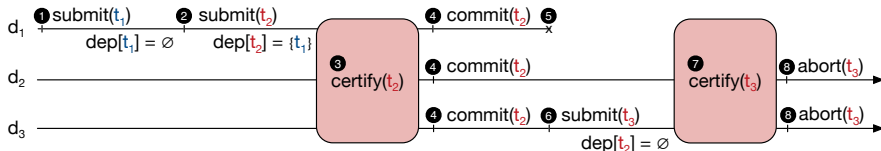
A strong transaction may have to **wait** for some of its dependencies to become uniform before committing.

However, this may cost too much.

# Performance of UNISTORE

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

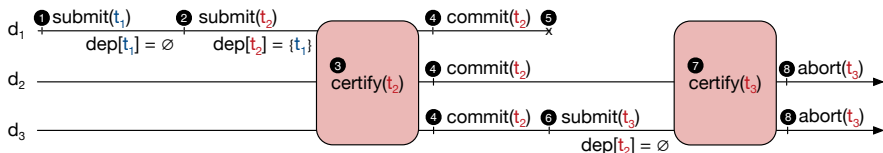
Causal transactions are executed on an (almost)  
**uniform snapshot** that may be slightly in the past.



# 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.

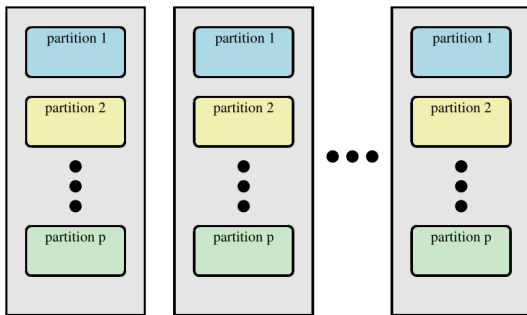
# Scalability of UNISTORE

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

# System Model

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

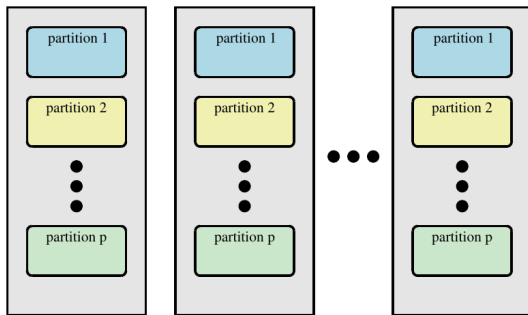
$\mathcal{P} = \{1, \dots, N\}$  : the set of (logical) partitions



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

# 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*.

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

For a transaction originating at data center  $d$ ,  
we call  $\textit{commitVec}[d]$  its **local timestamp**.

# Metadata for Causal Transactions

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

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

For a transaction originating at data center  $d$ ,  
we call  $\textit{commitVec}[d]$  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:

$$\text{knownVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

$$\text{stableVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

$$\text{uniformVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$



# Metadata for Causal Transactions

$$\text{knownVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

Property (Property of  $\text{knownVec}$ )

For each data center  $i$ ,

the **replica**  $p_d^m$  stores the updates to partition  $m$   
by transactions originating at  $i$  with local timestamps  $\leq \text{knownVec}[i]$ .

# Metadata for Causal Transactions

$$\text{stableVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

Property (Property of `stableVec`)

For each data center  $i$ ,  
the **data center  $d$**  stores the updates  
by transactions originating at  $i$  with local timestamps  $\leq \text{stableVec}[i]$ .

# Metadata for Causal Transactions

```
1: function BROADCAST_VECS()  
2:   send KNOWNVEC_LOCAL( $m, \text{knownVec}$ ) to  $p_d^l, l \in \mathcal{P}$   
3:   send STABLEVEC( $d, \text{stableVec}$ ) to  $p_i^m, i \in \mathcal{D}$   
4:   send KNOWNVEC_GLOBAL( $d, \text{knownVec}$ ) to  $p_i^m, i \in \mathcal{D}$ 
```

$$\text{stableVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

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

# Metadata for Causal Transactions

$$\text{uniformVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

## Property (Property of $\text{uniformVec}$ )

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

# Metadata for Causal Transactions

```
1: function BROADCAST_VECS()  
2:   send KNOWNVEC_LOCAL( $m$ , knownVec) to  $p_d^l, l \in \mathcal{P}$   
3:   send STABLEVEC( $d$ , stableVec) to  $p_i^m, i \in \mathcal{D}$   
4:   send KNOWNVEC_GLOBAL( $d$ , knownVec) to  $p_i^m, i \in \mathcal{D}$ 
```

$$\text{uniformVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

```
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  $j \in \mathcal{D}$  do  
14:     var  $ts \leftarrow \max\{\min\{\text{stableMatrix}[h][j] \mid h \in g\} \mid g \in G\}$   
15:     uniformVec[ $j$ ]  $\leftarrow \max\{\text{uniformVec}[j], ts\}$ 
```

# Metadata for Causal Transactions

$$\text{uniformVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

## Lemma

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

# Metadata for Causal Transactions

$$\text{uniformVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

## 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

$\text{pastVec}$  : causal past of client

- 1: **function** START()
- 2:      $p \leftarrow$  a random partition in data center  $d$
- 3:      $\langle \text{tid}, \text{snapVec} \rangle \leftarrow$  **send** START\_TX( $\text{pastVec}$ ) **to**  $p$
- 4:      $\text{pastVec} \leftarrow \text{snapVec}$
- 5:     **return**  $\text{tid}$

$\forall i \in \mathcal{D} \setminus \{d\}$ , all transactions originating at  $i$   
with local timestamps  $\leq \text{pastVec}[i]$  are already uniform.



# Causal Consistency Protocol: Start

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

```
1: function START_TX( $V$ )
2:   for  $i \in \mathcal{D} \setminus \{d\}$  do
3:      $\text{uniformVec}[i] \leftarrow \max\{V[i], \text{uniformVec}[i]\}$ 
4:   var  $tid \leftarrow \text{generate\_tid}()$ 
5:    $\text{snapVec}[tid] \leftarrow \text{uniformVec}$ 
6:    $\text{snapVec}[tid][d] \leftarrow \max\{V[d], \text{uniformVec}[d]\}$ 
7:    $\text{snapVec}[tid][\text{strong}] \leftarrow \max\{V[\text{strong}], \text{stableVec}[\text{strong}]\}$ 
8:   return  $\langle tid, \text{snapVec}[tid] \rangle$ 
```

$\text{snapVec}[tid][d]$  ensures “read-your-writes”.

# Causal Consistency Protocol: Update

```
11: function UPDATE( $k, v$ )
12:    $_ \leftarrow \text{send DO\_UPDATE}(tid, k, v) \text{ to } p$ 
13:   return ok
```

```
17: function DO_UPDATE( $tid, k, v$ )
18:   var  $l \leftarrow \text{partition}(k)$ 
19:    $\text{wbuff}[tid][l][k] \leftarrow v$ 
20:    $\text{rset}[tid][l] \leftarrow \text{rset}[tid][l] \cup \{k\}$ 
21:   return ok
```

$\text{wbuff}[tid][l]$  : buffer for the latest local update on each key

# Causal Consistency Protocol: Read

```
6: function READ( $k$ )
7:    $\langle v, c \rangle \leftarrow \text{send DO\_READ}(\text{tid}, k, \text{lc})$  to  $p$ 
8:   if  $c \neq \perp$  then
9:      $\text{lc} \leftarrow \max\{\text{lc}, c\}$ 
10:  return  $v$ 
```

```
9: function DO_READ( $\text{tid}, k, c$ )
10:   $\text{lc} \leftarrow \max\{\text{lc}, c\}$ 
11:  var  $l \leftarrow \text{partition}(k)$ 
12:  if  $\text{wbuff}[\text{tid}][l][k] \neq \perp$  then
13:    return  $\langle \text{wbuff}[\text{tid}][l][k], \perp \rangle$ 
14:   $\langle v, c \rangle \leftarrow \text{send READ\_KEY}(\text{snapVec}[\text{tid}], k)$  to  $p_d^l$ 
15:   $\text{rset}[\text{tid}][l] \leftarrow \text{rset}[\text{tid}][l] \cup \{k\}$ 
16:  return  $\langle v, c \rangle$ 
```

# Causal Consistency Protocol: Read

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

- 1: **when received** `READ_KEY(snapVec, k)` **from** *p*
- 2:   **for**  $i \in \mathcal{D} \setminus \{d\}$  **do**
- 3:     `uniformVec[i]  $\leftarrow$  max{snapVec[i], uniformVec[i]}`
- 4:   **wait until** `knownVec[d]  $\geq$  snapVec[d]  $\wedge$  knownVec[strong]  $\geq$  snapVec[strong]`
- 5:    $\langle v, \text{commitVec}, c \rangle \leftarrow \text{snapshot}(\text{opLog}[k], \text{snapVec})$   $\triangleright$  returns the latest *commitVec* (in terms of Lamport clock order in Definition 50) such that *commitVec*  $\leq$  *snapVec*
- 6:   **send**  $\langle v, c \rangle$  **to** *p*

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

# Causal Consistency Protocol: Commit

```
14: function COMMIT_CAUSAL_TX()  
15:    $\langle vc, c \rangle \leftarrow \text{send COMMIT\_CAUSAL}(\text{tid}, lc) \text{ to } p$   
16:    $\text{pastVec} \leftarrow vc$   
17:    $lc \leftarrow c$   
18:   return ok
```

# Causal Consistency Protocol: Commit

Read-only transactions returns immediately.

```
22: function COMMIT_CAUSAL( $tid, c$ )
23:    $lc \leftarrow \max\{lc, c\} + 1$ 
24:   if  $\forall l \in \mathcal{P}. wbuff[tid][l] = \emptyset$  then
25:     return  $\langle snapVec[tid], lc \rangle$ 
26:   var  $commitVec \leftarrow snapVec[tid]$ 
27:   send PREPARE( $tid, wbuff[tid][l], snapVec[tid]$ ) to  $p_d^l, l \in \mathcal{P}$ 
28:   for all  $l \in \mathcal{P}$  do
29:     wait receive PREPARE_ACK( $tid, ts$ ) from  $p_d^l$ 
30:      $commitVec[d] \leftarrow \max\{commitVec[d], ts\}$ 
31:   send COMMIT( $tid, commitVec, lc$ ) to  $p_d^l, l \in \mathcal{P}$ 
32:   return  $\langle commitVec, lc \rangle$ 
```

2PC protocol for update transactions

# Causal Consistency Protocol: Commit

$ts$  : **prepare timestamp** from its local clock

```
7: when received PREPARE( $tid, wbuff, snapVec$ ) from  $p$ 
8:   for  $i \in \mathcal{D} \setminus \{d\}$  do
9:      $uniformVec[i] \leftarrow \max\{snapVec[i], uniformVec[i]\}$ 
10:  var  $ts \leftarrow \text{clock}$ 
11:   $preparedCausal \leftarrow preparedCausal \cup \{\langle tid, wbuff, ts \rangle\}$ 
12:  send PREPARE_ACK( $tid, ts$ ) to  $p$ 
```

# Causal Consistency Protocol: Commit

**wait** : ensure that its local clock is up-to-date

```
13: when received COMMIT( $tid, commitVec, c$ )  
14:   wait until clock  $\geq commitVec[d]$   
15:    $\langle tid, wbuff, - \rangle \leftarrow \text{find}(tid, preparedCausal)$   
16:    $preparedCausal \leftarrow preparedCausal \setminus \{ \langle tid, -, - \rangle \}$   
17:   for all  $\langle k, v \rangle \in wbuff$  do  
18:      $opLog[k] \leftarrow opLog[k] \cdot \langle v, commitVec, c \rangle$   
19:    $committedCausal[d] \leftarrow committedCausal[d] \cup \{ \langle tid, wbuff, commitVec, c \rangle \}$ 
```

$committedCausal[d]$  : for replication



# Causal Consistency Protocol: Replication

## 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  $\leq \text{knownVec}[i]$ .

```
1: function PROPAGATE_LOCAL_TXS()
2:   if preparedCausal =  $\emptyset$  then
3:     knownVec[d]  $\leftarrow$  clock
4:   else
5:     knownVec[d]  $\leftarrow$  min{ts |  $\langle -, -, ts \rangle \in \text{preparedCausal}$ } - 1
6:   var txs  $\leftarrow$  { $\langle -, -, \text{commitVec}, c \rangle \in \text{committedCausal}[d] \mid \text{commitVec}[d] \leq \text{knownVec}[d]$ }
7:   if txs  $\neq \emptyset$  then
8:     send REPLICATE( $d, \text{txs}$ ) to  $p_i^m, i \in \mathcal{D} \setminus \{d\}$ 
9:     committedCausal[d]  $\leftarrow$  committedCausal[d]  $\setminus$  txs
10:  else
11:    send HEARTBEAT( $d, \text{knownVec}[d]$ ) to  $p_i^m, i \in \mathcal{D} \setminus \{d\}$ 
```

HEARTBEAT : for liveness

# 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**.

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

# Metadata for Strong Transactions

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

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

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

# Metadata for Strong Transactions

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

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

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

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

# Metadata for Strong Transactions

$$\text{uniformVec} \in [\mathcal{D} \rightarrow \mathbb{N}]$$

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

The commit protocol for strong transactions  
guarantees their uniformity.

# Strong Consistency Protocol: Commit

```
1: function COMMIT_STRONG( $tid, c$ )
2:   UNIFORM_BARRIER( $\text{snapVec}[tid]$ )
3:    $\langle d, vc, c \rangle \leftarrow \text{CERTIFY } tid, \text{wbuff}[tid], \text{rset}[tid], \text{snapVec}[tid], c$ 
4:    $lc \leftarrow \max\{lc, c\} + 1$ 
5:   return  $\langle d, vc, lc \rangle$ 
```

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:    $lc \leftarrow \max\{lc, c\} + 1$ 
22:   wait until  $\text{uniformVec}[d] \geq V[d]$ 
23:   return  $lc$ 
```

# Strong Consistency Protocol: Commit

```
1: function COMMIT_STRONG(tid, c)  
2:   UNIFORM_BARRIER(snapVec[tid])  
3:    $\langle d, vc, c \rangle \leftarrow \text{CERTIFY}(tid, wbuff[tid], rset[tid], snapVec[tid], c)$   
4:    $lc \leftarrow \max\{lc, c\} + 1$   
5:   return  $\langle d, vc, lc \rangle$ 
```

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



# Strong Consistency Protocol: Commit

```
1: function COMMIT_STRONG(tid, c)
2:   UNIFORM_BARRIER(snapVec[tid])
3:    $\langle d, vc, c \rangle \leftarrow \text{CERTIFY}(tid, wbuff[tid], rset[tid], snapVec[tid], c)$ 
4:    $lc \leftarrow \max\{lc, c\} + 1$ 
5:   return  $\langle d, vc, lc \rangle$ 
```

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

## Multi-Shot Distributed Transaction Commit

Gregory Chockler

Royal Holloway, University of London, UK

Alexey Gotsman<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

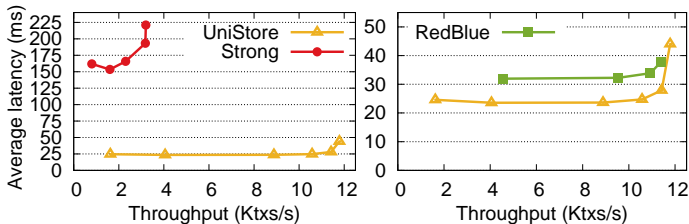
# Strong Consistency Protocol: Deliver

```
6: upon DELIVER_UPDATES( $W$ )  
7:   for  $\langle k, v, \text{commitVec}, c \rangle \in W$  in  $\text{commitVec}[\text{strong}]$  order do  
8:      $\text{opLog}[k] \leftarrow \text{opLog}[k] \cdot \langle v, \text{commitVec}, c \rangle$   
9:      $\text{knownVec}[\text{strong}] \leftarrow \text{commitVec}[\text{strong}]$ 
```

# Evaluation

# Performance of UNiSTORE

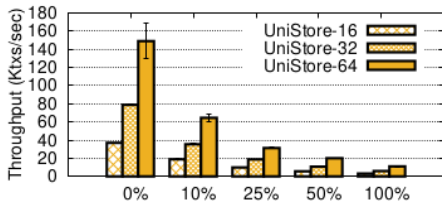
Throughput: 5% and 259% higher than REDBLUE and STRONG



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)