Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp

Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, Binyu Zang
Institute of Parallel and Distributed Systems@Shanghai Jiao Tong University

Multi-versioning is important for TXs[1]

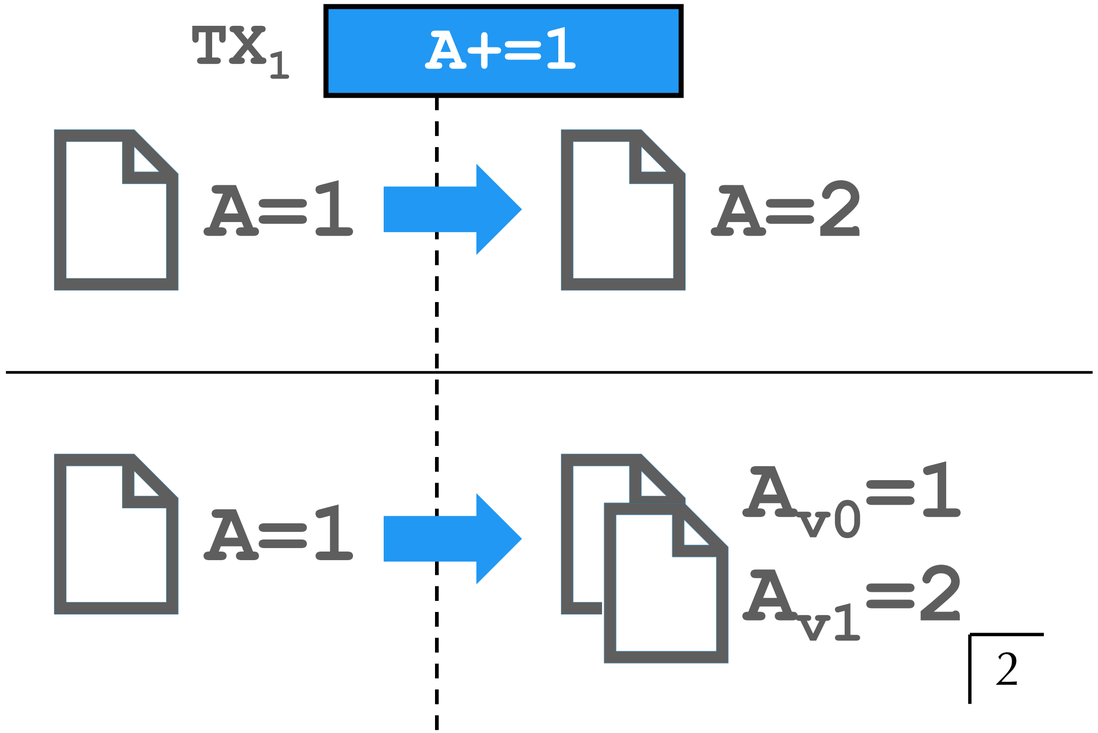
Data are stored in **multiple-copies** (aka, versions)

- e.g., updates **not overwrite** (as in single-versioning)

Single-version database

vs.

Multi-version (MV) database



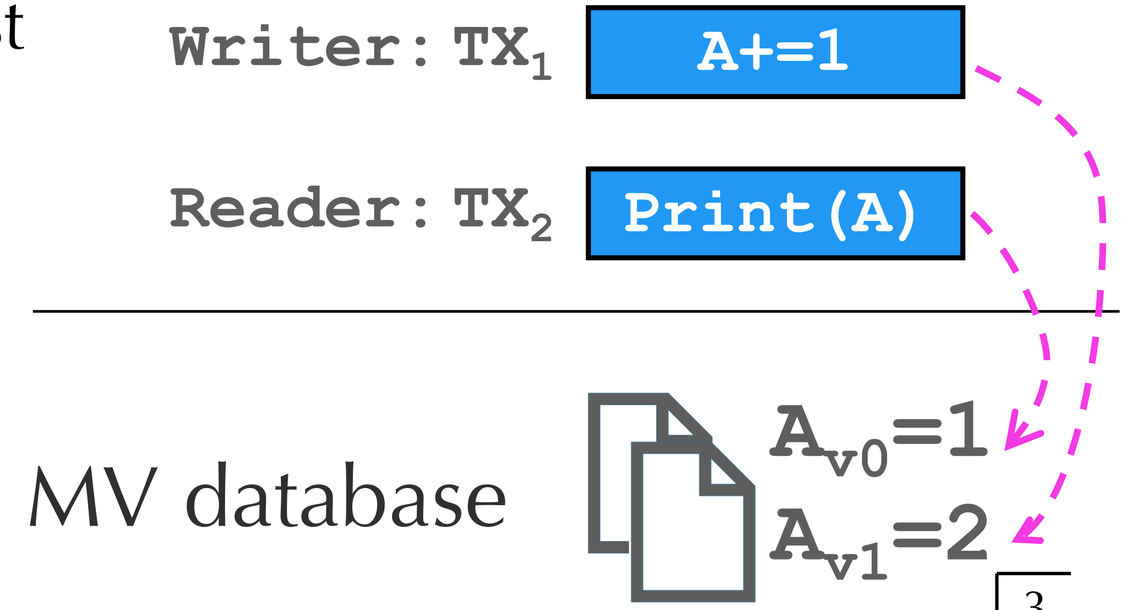
[1] Transactions

Multi-versioning is important for TXs[1]

Data are stored in **multiple-copies** (aka, versions)

Higher concurrency for readers writers

- Reader reads from the (consistent) past
- Write **concurrently** updates the latest



[1] Transactions

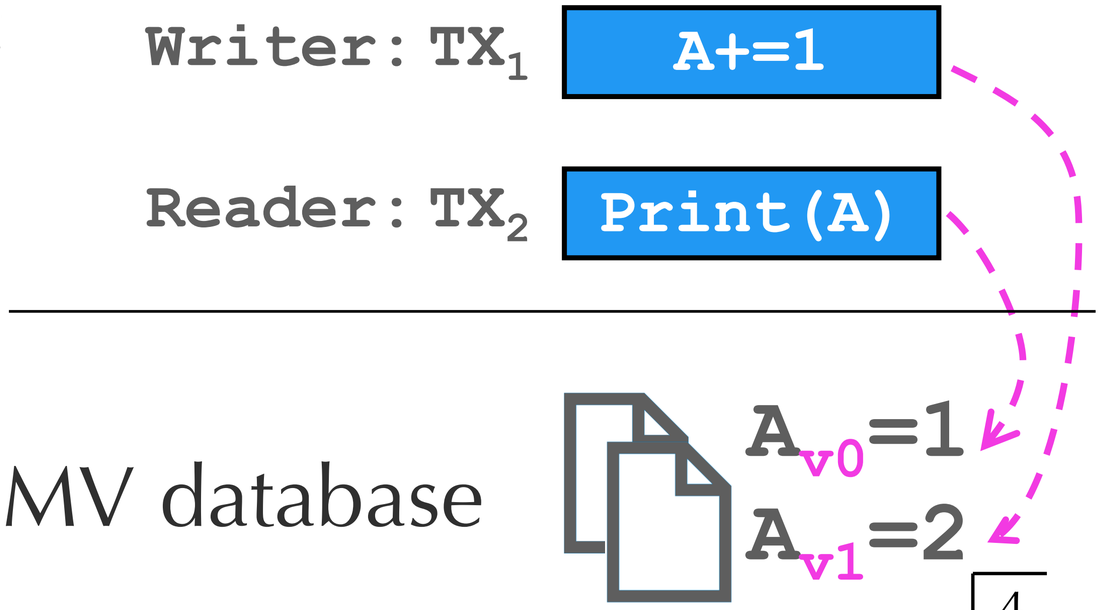
Multi-versioning is important for TXs[1]

Data are stored in **multiple-copies** (aka, versions)

Higher concurrency for readers writers

- Reader reads from the (consistent) past
- Write **concurrently** updates the latest

How to assign version
& choose version to read?



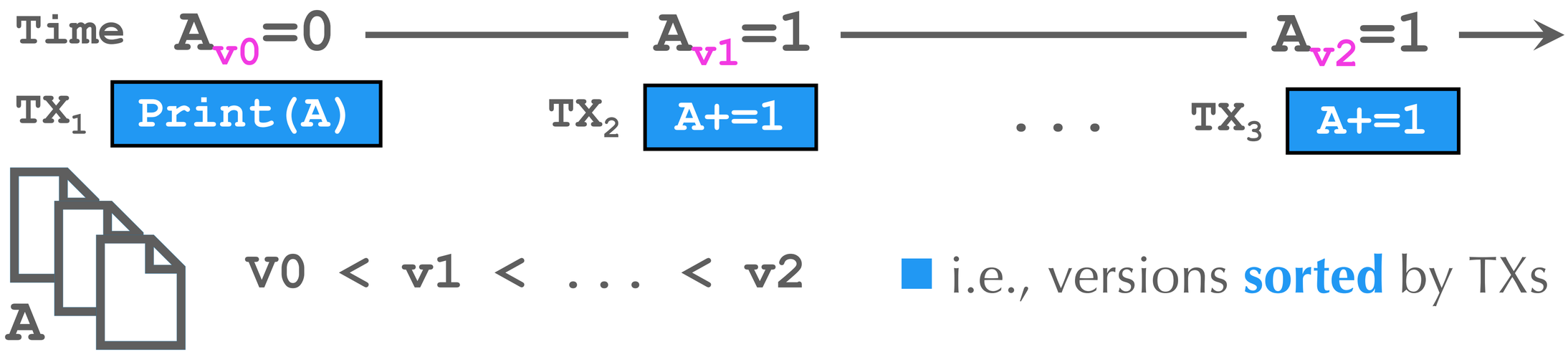
[1] Transactions

How to assign version & choose version to read?

How to assign version & choose version to read?

Deciding version should be **consistent & efficient**

- **Consistent** = Match TX's order



v0 < v1 < . . . < v2

- i.e., versions **sorted** by TXs

How to assign version & choose version to read?

Deciding version should be **consistent & efficient**

- **Consistent** = Match TX's order
- **Efficient**: minimal impact to the TX

A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall **the goals: Consistency + Efficiency**

A common approach: use timestamp

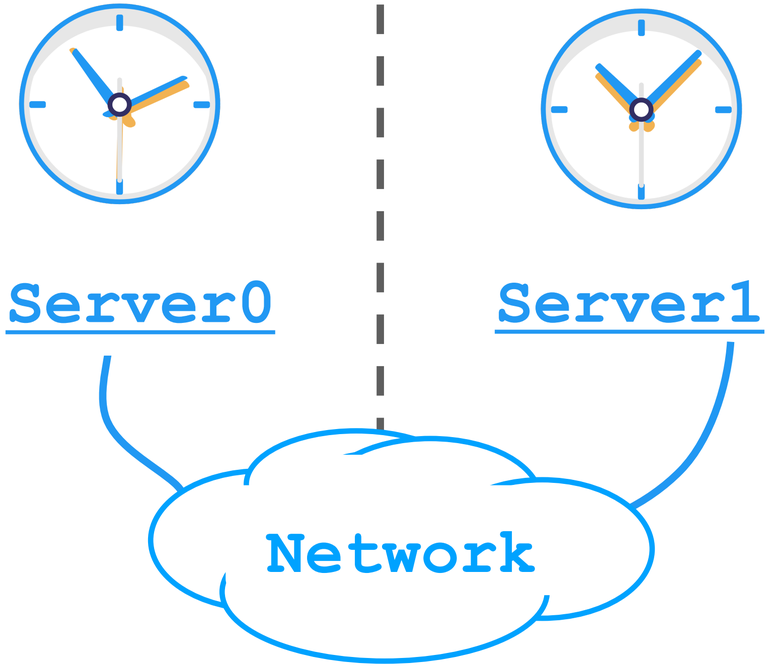
Yet, ordering timestamps is **challenging in networked systems**

- Recall **the goals: Consistency + Efficiency**

#1. Physical clock

✓ Nearly **no cost**

✗ Not consistent: **unsynchronized clocks**



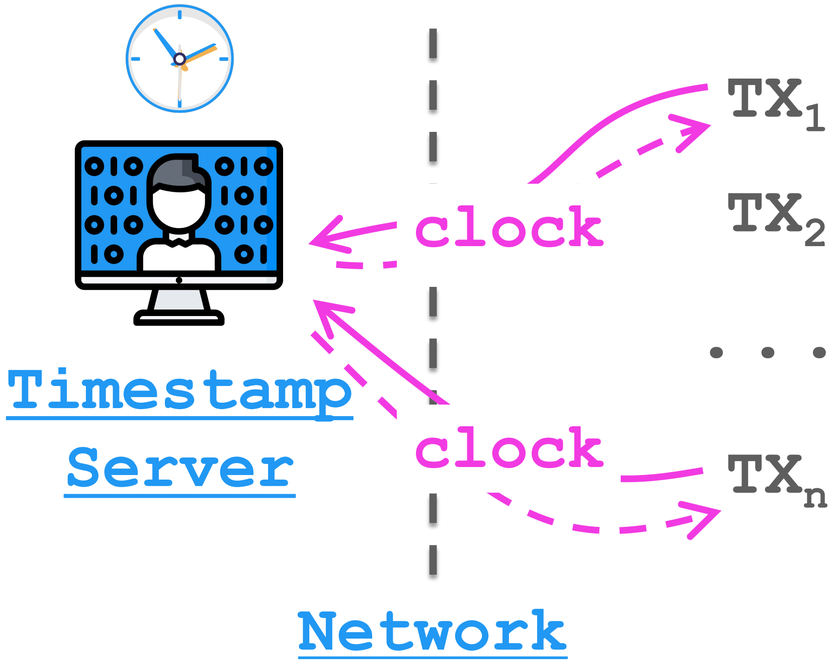
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall **the goals: Consistency + Efficiency**

#2. Global logical clock **(GTS)**

✓ Easy to ensure consistency



A common approach: use timestamp

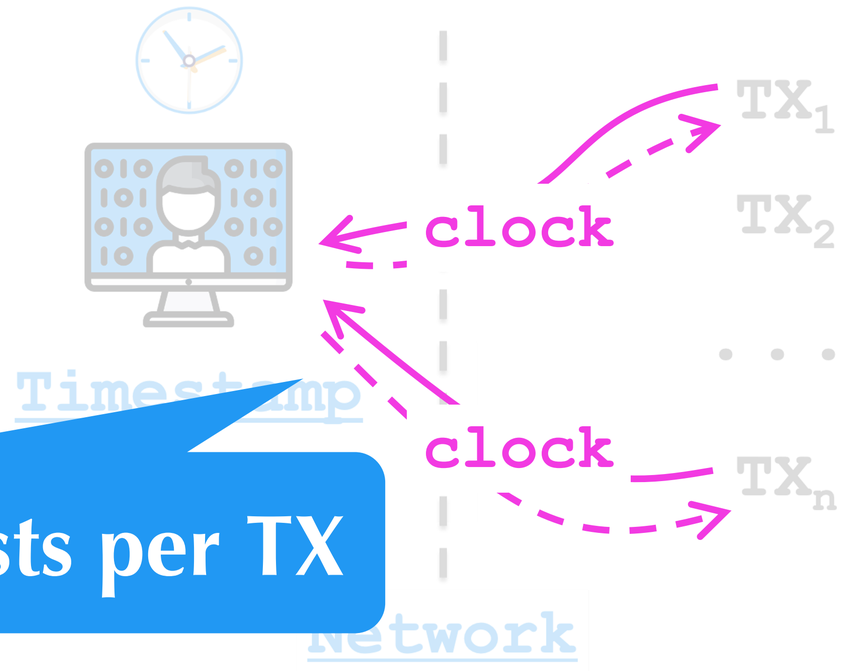
Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#2. Global logical clock **(GTS)**

✓ Easy to ensure consistency

✗ **Non-trivial costs**



Extra network requests per TX

A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

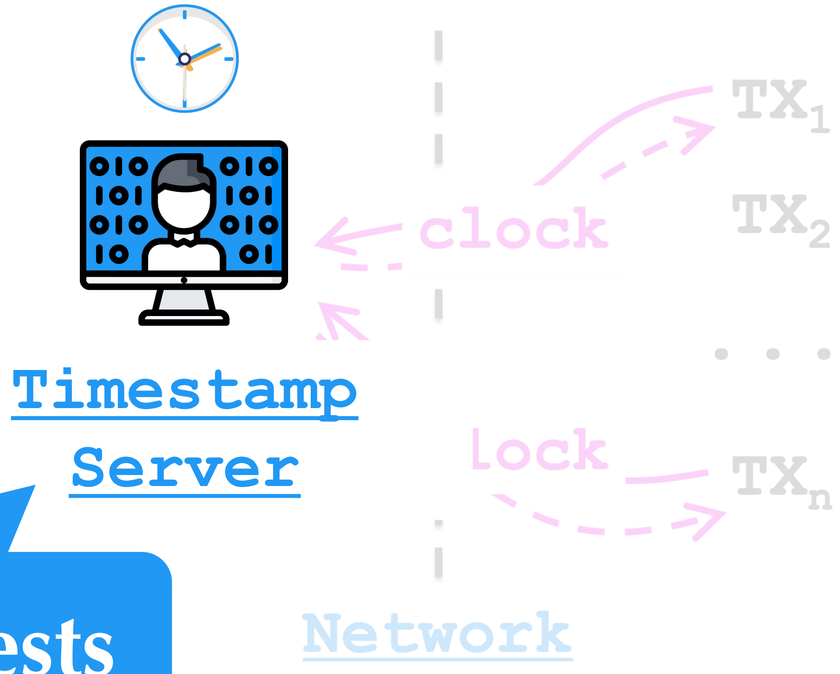
- Recall the goals: Consistency + Efficiency

#2. Global logical clock **(GTS)**

✓ Easy to ensure consistency

✗ Non-trivial costs

✗ **Scalability bottleneck**



Handles all the requests

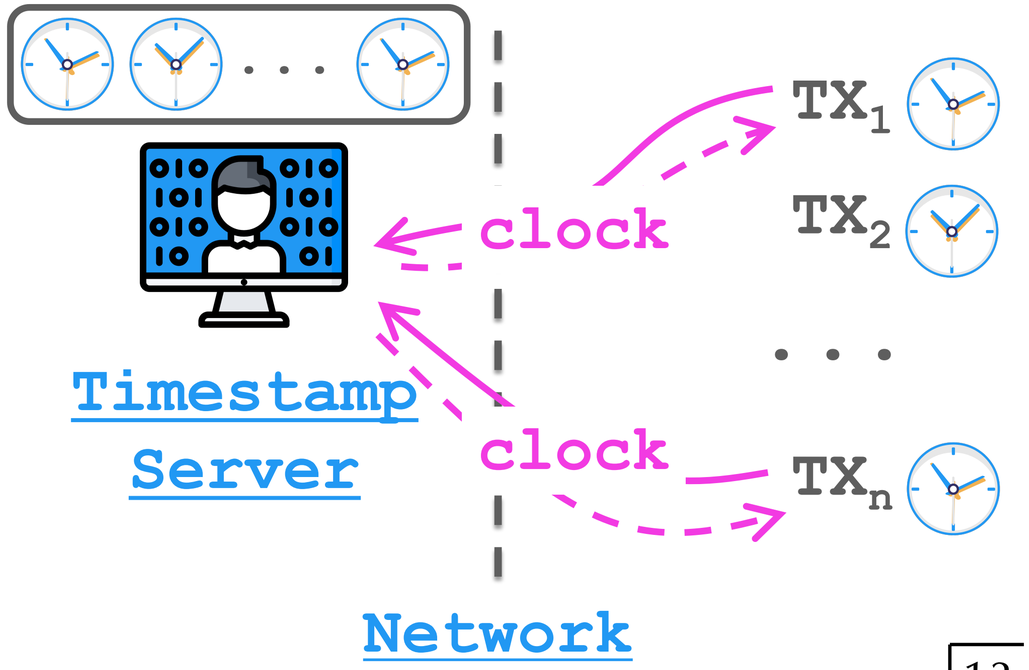
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#3. Global **vectorized** clock **(VTS)**

- Per-worker clock



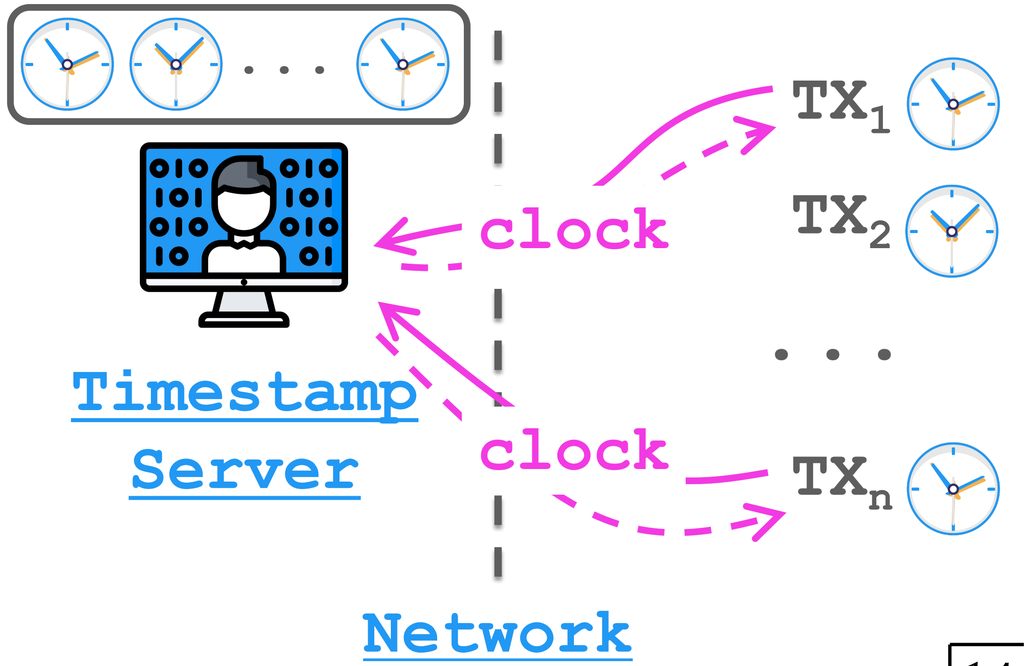
A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#3. Global **vectorized** clock **(VTS)**

✓ Reduce requests to the global server



A common approach: use timestamp

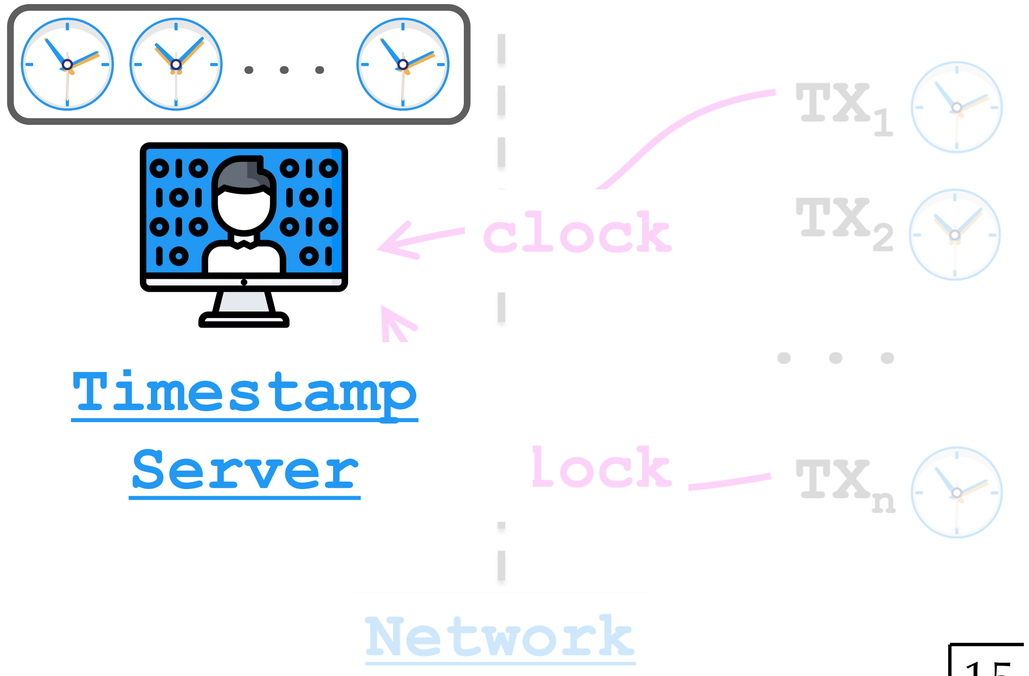
Yet, ordering timestamps is **challenging in networked systems**

- Recall the goals: Consistency + Efficiency

#3. Global **vectorized** clock **(VTS)**

✓ Reduce requests to the global server

✗ Still needs the **global server**



A common approach: use timestamp

Yet, ordering timestamps is **challenging in networked systems**

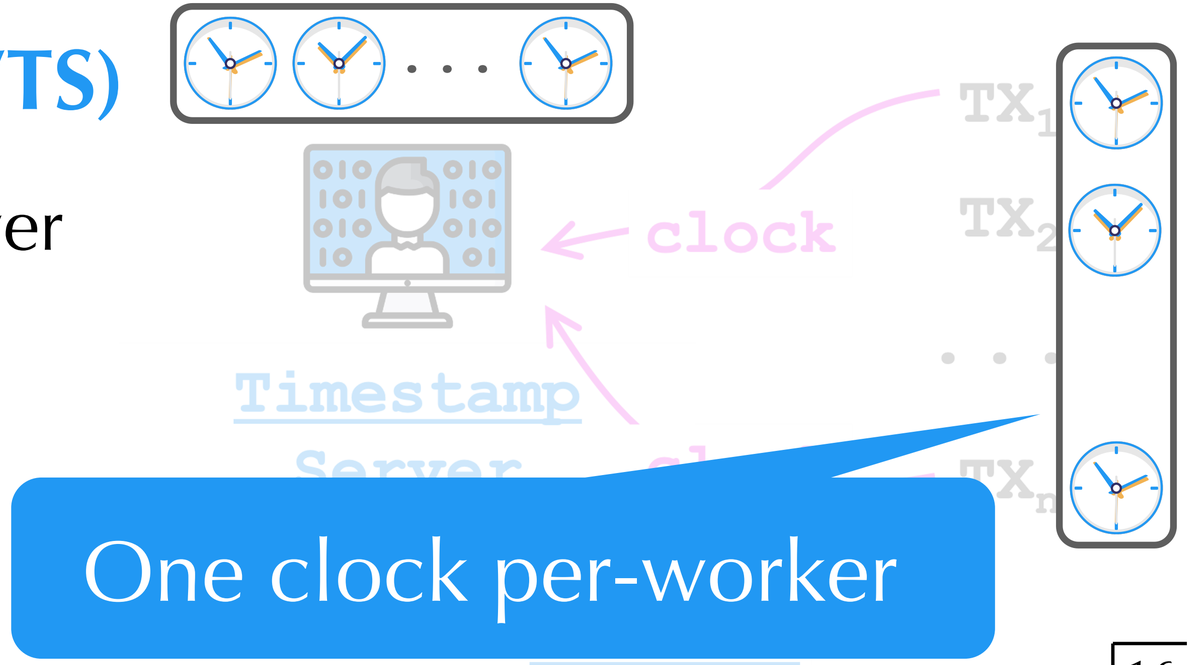
- Recall the goals: Consistency + Efficiency

#3. Global **vectorized** clock **(VTS)**

✓ Reduce requests to the global server

✗ Still needs the global server

✗ **Non-scalable** meta-data



Can we make the timestamps ordered,

with (nearly) zero cost?

Observation: concurrency control (CC)

Recall: consistency = **timestamps match TXs' order**

- While **CC orders TXs** (e.g., 2PL OCC)

Physical clocks are approximately good

- **Hybrid clock**: using **CC in read-write TX** to order physical clocks

This work: Decentralized Scalar Timestamp

Scalable scalar timestamp mechanism to enable **MVCC**

- Provide **snapshot reads** for read-only TX (i.e., bypasses original CC)

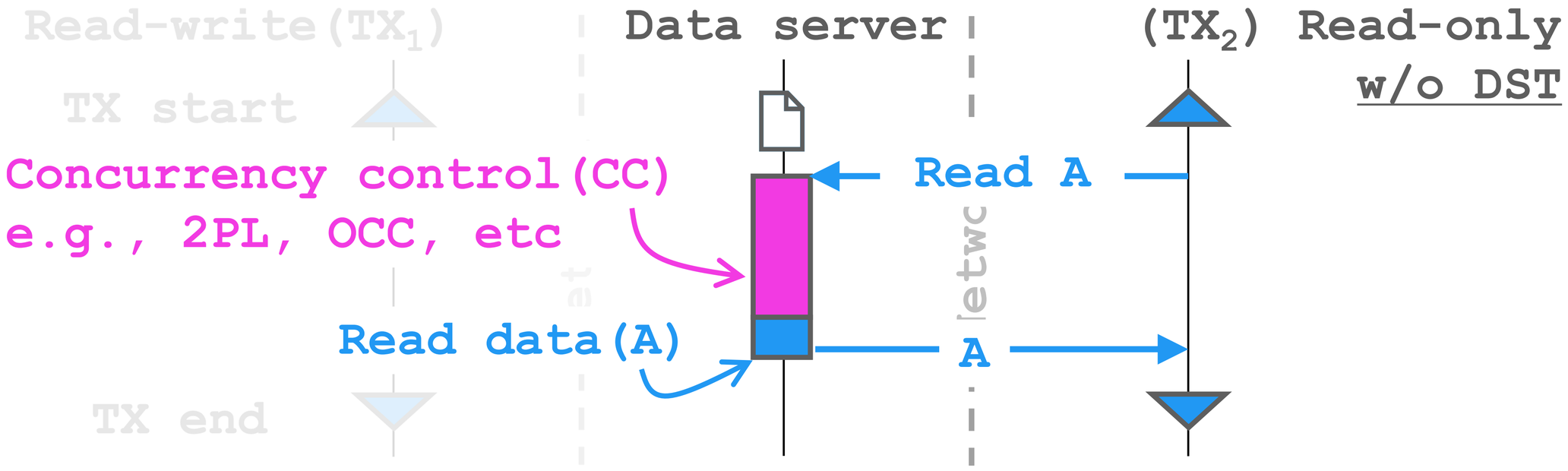
Consistent & efficient scalar hybrid physical-logical clock

- Reader uses the physical clock to read snapshots (bypass CC)
- DST is consistent by piggybacking w/ read-write TX's CC

This work: Decentralized Scalar Timestamp

Scalable scalar timestamp mechanism to enable **MVCC**

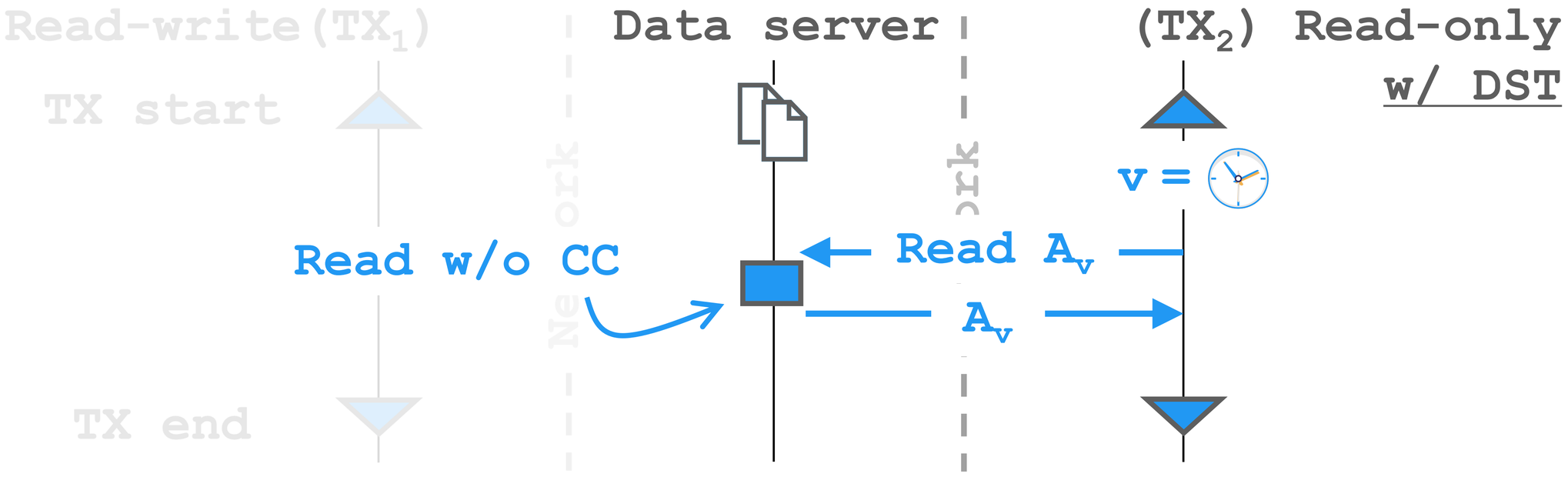
- Provide **snapshot reads** for read-only TX (i.e., bypasses original CC)



This work: Decentralized Scalar Timestamp

Scalable scalar timestamp mechanism to enable **MVCC**

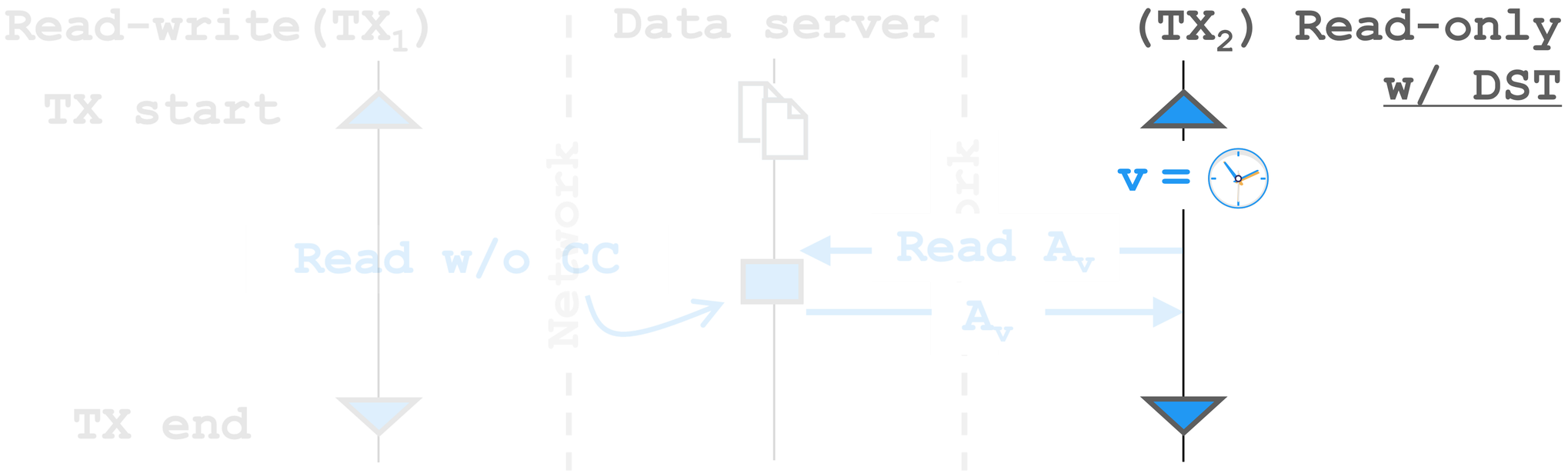
- Provide **snapshot reads** for read-only TX (i.e., bypasses original CC)



This work: Decentralized Scalar Timestamp

Use physical clock as an tentative timestamp

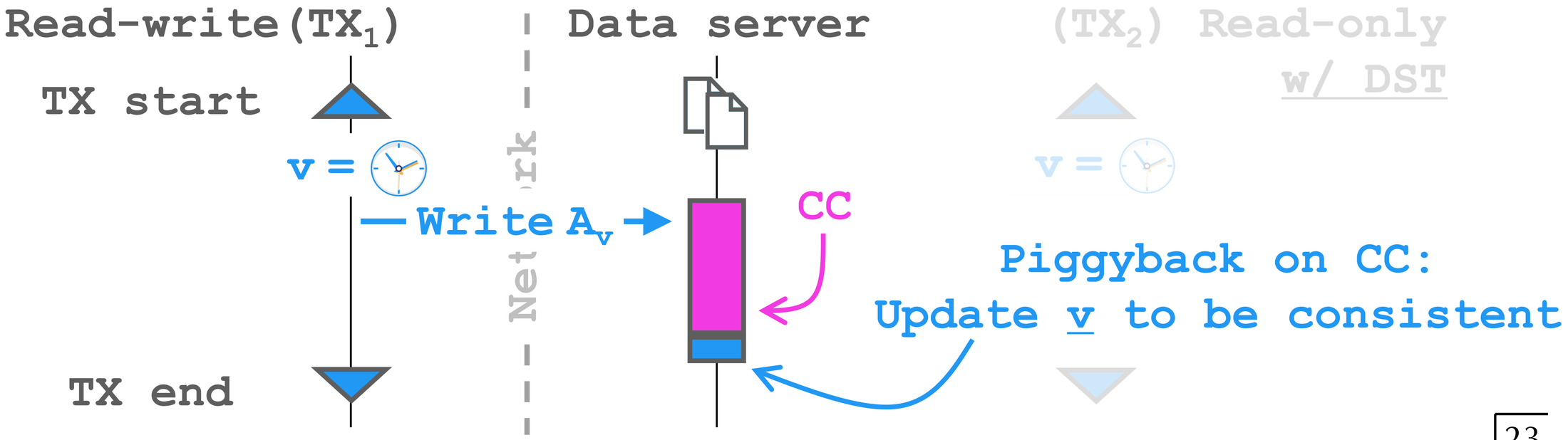
- **Minimal cost** & **Scalable**: scalar + no global timestamp



This work: Decentralized Scalar Timestamp

Use the **(un-sync)** physical is not sufficient for read-write TXs

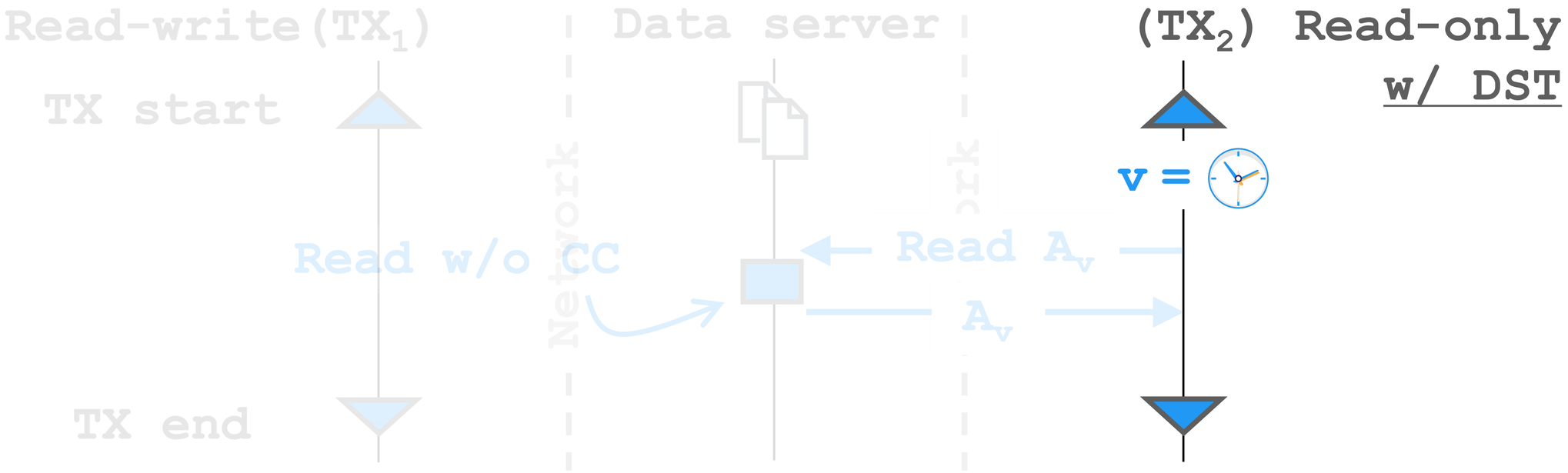
- DST is consistent by piggybacking w/ read-write TX's CC



This work: Decentralized Scalar Timestamp

Use physical clock as an tentative timestamp

- Also **fresh**: snapshot has **bounded staleness** (e.g., ms-scale)



Decentralized (Scalar) Timestamp is not so new

Based on many prior work on **decentralized timestamps**

- **TicToc**@SIGMOD'18, **Cicada**@SIGMOD'17, etc

Differently:

- DST is a general timestamp method for **different CC**
- DST further targets **MVCC in a distributed setting**

Outline of the remaining content

How to piggyback on CC for read-write TX ? DST rules

How snapshot reads bypass CC? DST reads

How good is DST's snapshot read? Bounded staleness

Evaluation results

Outline of the remaining content

How to piggyback on CC for read-write TX ? **DST rules**

How snapshot reads bypass CC ? DST reads

How good is DST's snapshot read? Bounded-staleness

Evaluations results

DST rules for read-write TX

Inspired from **conflicting relationships** of TXs

Two TXs conflict if and only if:

TX access	Read	Read
Read	-	Read-Write
Write	Write-Read	Write-Write

DST rules for read-write TX

Inspired from **conflicting relationships** of TXs

Two TXs conflict if and only if:

TX access	**Read**	**Read**
Read	-	Read-Write
Write	Write-Read	Write-Write

Timestamp should adjust **follow these rules**, too !

Write-Write

If TX_2 has a write-write conflict with TX_1

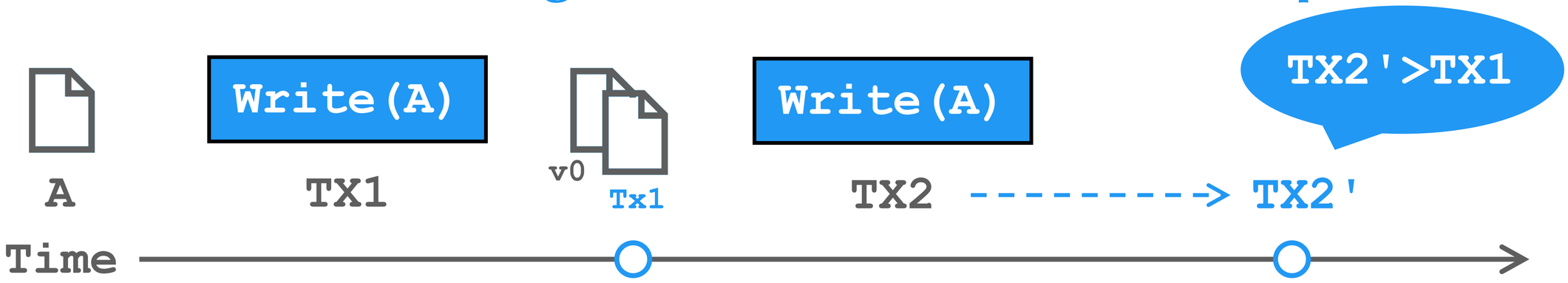
- $Timestamp_{tx2}$(Tx2) should > $Timestamp_{tx1}$ (Tx1)

Write-Write

If TX_2 has a write-write conflict with TX_1

- $Timestamp_{tx2}$(Tx2) should > $Timestamp_{tx1}$ (Tx1)

Achieved via **storing the (last-writer) timestamp with data**

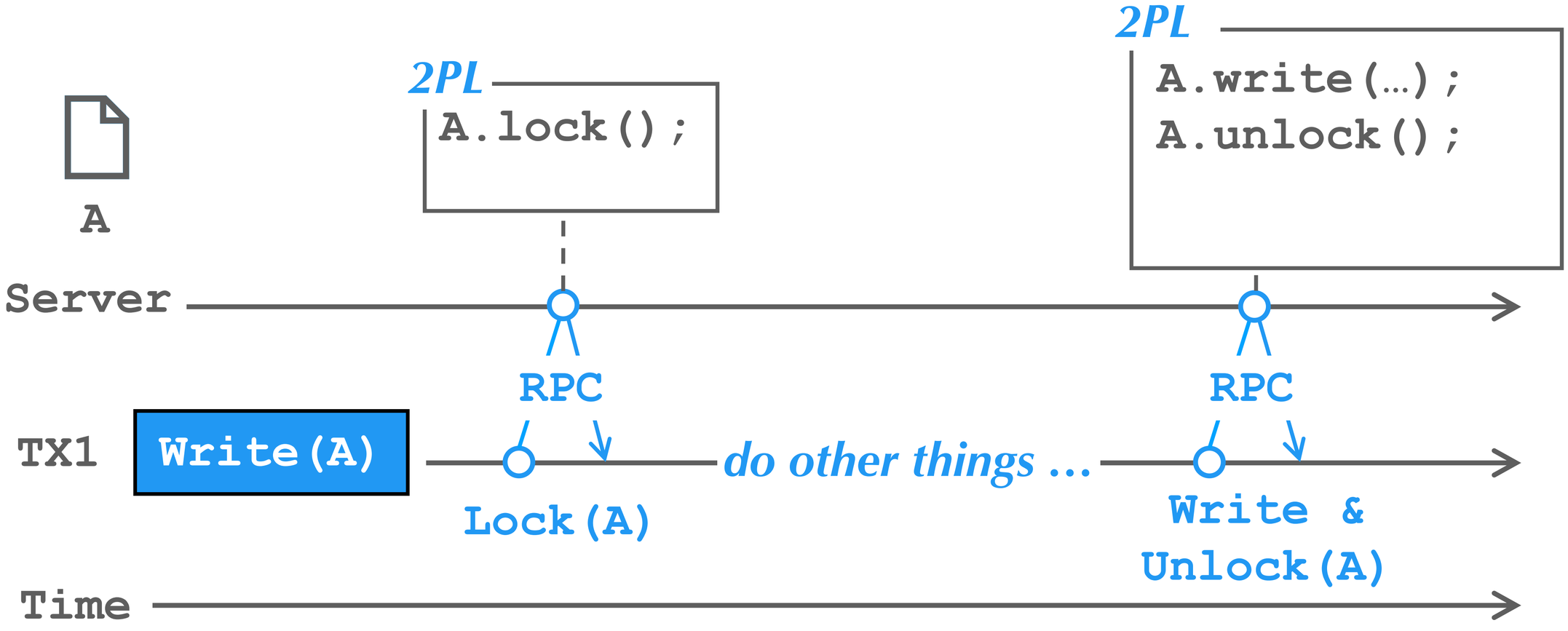


Write-Write rule#1:
A.ver = TX1

Write-Write rule#2:
TX'= MAX(TX,A.ver)+ 1

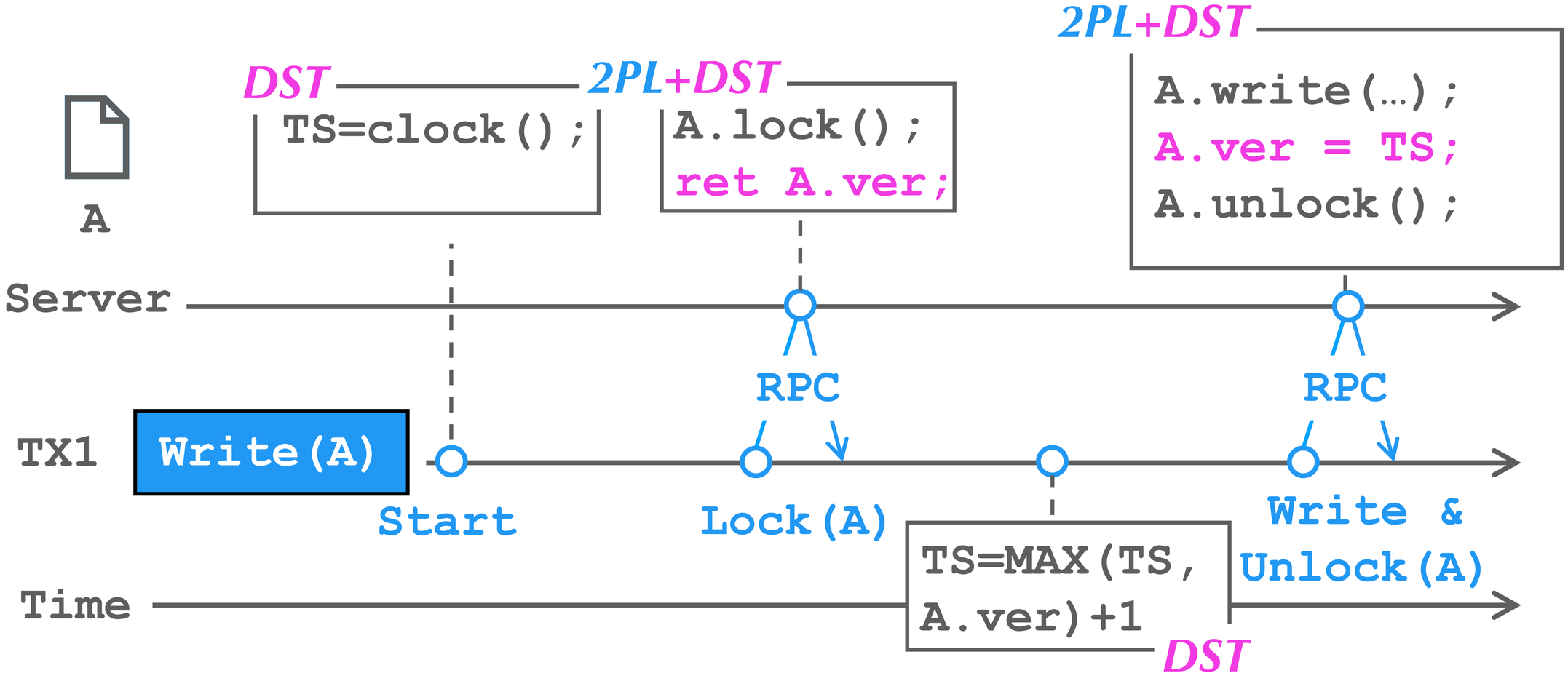
Example: Write-Write in 2PL

Can be simply piggybacked to 2PL in a lightweight way



Example: Write-Write in 2PL

Can be simply piggybacked to 2PL in a lightweight way



DST rules for read-write TX

Inspired from **conflicting relationships** of TXs

Two TXs conflict if and only if:

Similar to Write-Write

TX access	Read	Read
Read	-	Read-Write
Write	Write-Read	Write-Write

Timestamp should adjust **follow these rules**, too !

Outline of the remaining content

How to piggyback on CC for read-write TX ? DST rules

How snapshot reads bypass CC? DST reads

How good is DST's snapshot read? Bounded staleness

Evaluation results

DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

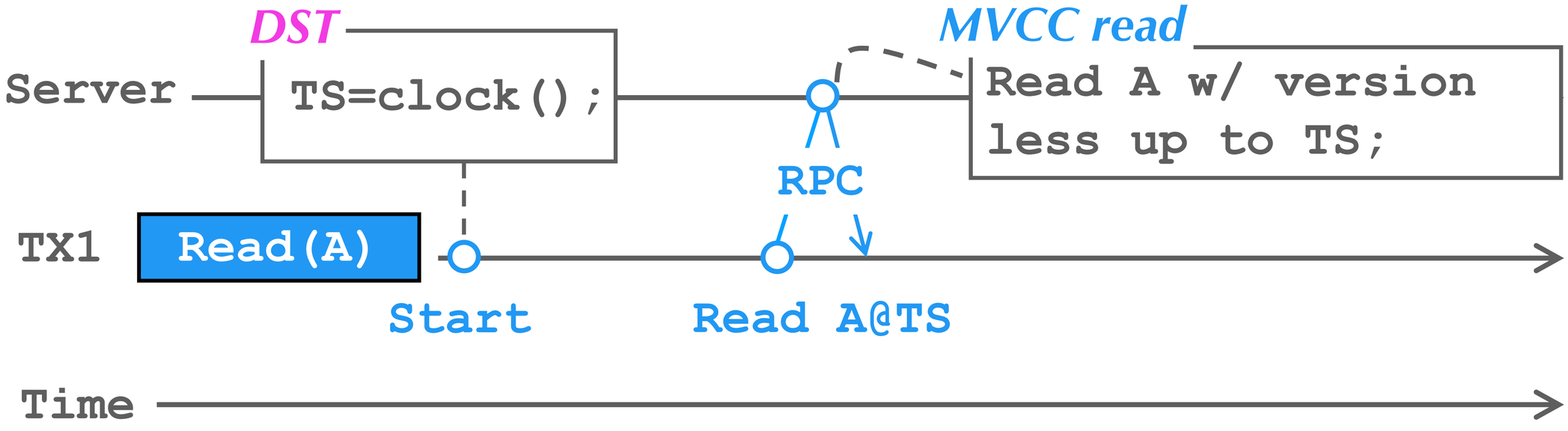
Can we use MVCC w/ DST the same as global clock?

DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

Can we use MVCC w/ DST the same as global clock?



DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

Can we use MVCC w/ DST the same as global clock?

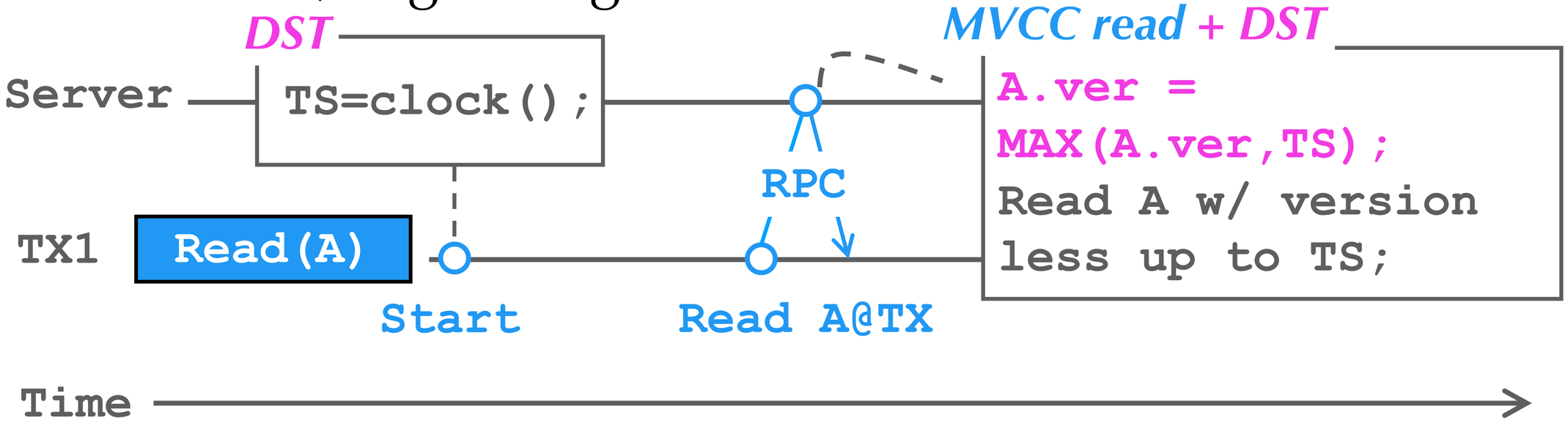
- **No**, because timestamp of reader does not use CC !
- i.e., not apply DST rules because rules are piggyback to the CC !

DST reads

Key benefit: Bypass CC for the reader

- e.g., avoid read locks in 2PL

DST reads (a lightweight rule for snapshot) to fix this:



Outline of the remaining content

How to piggyback on CC for read-write TX ? DST rules

How snapshot reads bypass CC? DST reads

How good is DST's snapshot read? Bounded staleness

Evaluation results

Evaluation results of DST

DST works well various CCs

- 2PL ⇨ MySQL cluster X DST
- OCC ⇨ DrTM+R[Eurosys'16] X DST
- ROCOCO ⇨ ROCOCO[OSDI'14] X DST

Also on various network settings

- Local clusters (TCP/IP), AWS cloud, RDMA

Evaluation results of DST

DST works well various CCs

- 2PL ⇨ MySQL cluster X DST
- OCC ⇨ DrTM+R[Eurosys'16] X DST
- ROCOCO ⇨ ROCOCO[OSDI'14] X DST

Also on various network settings

- Local clusters (TCP/IP), AWS cloud, RDMA

2PL X DST

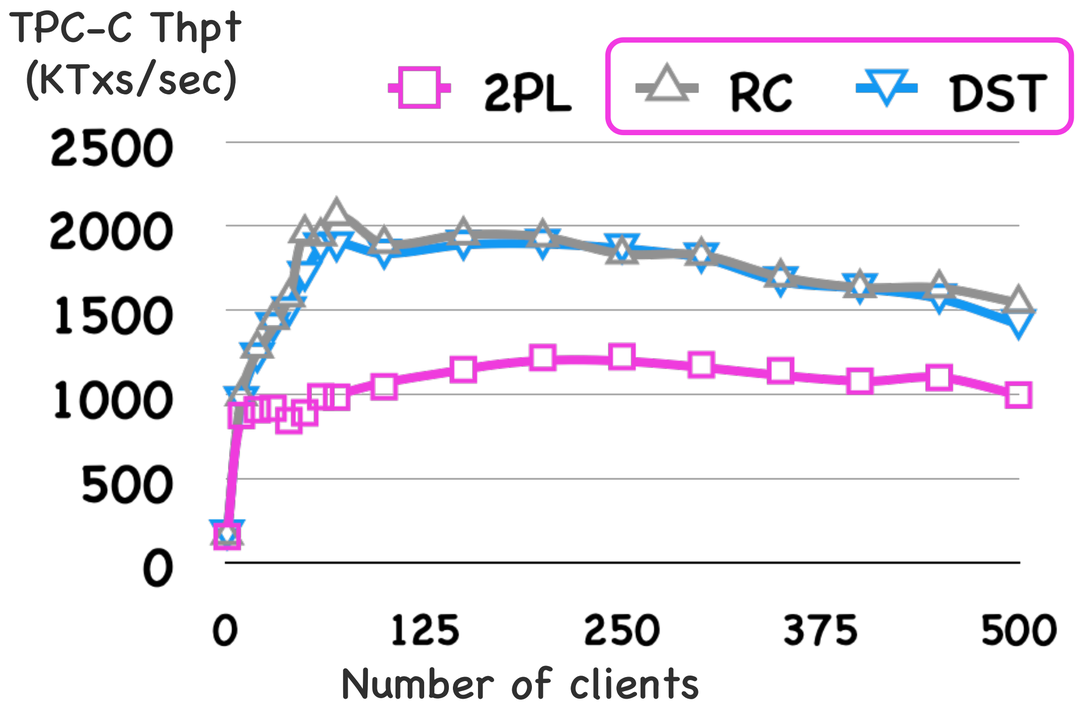
Cluster Name	CPU	Network	Num
Val	2 X 12 cores	10GbE	16

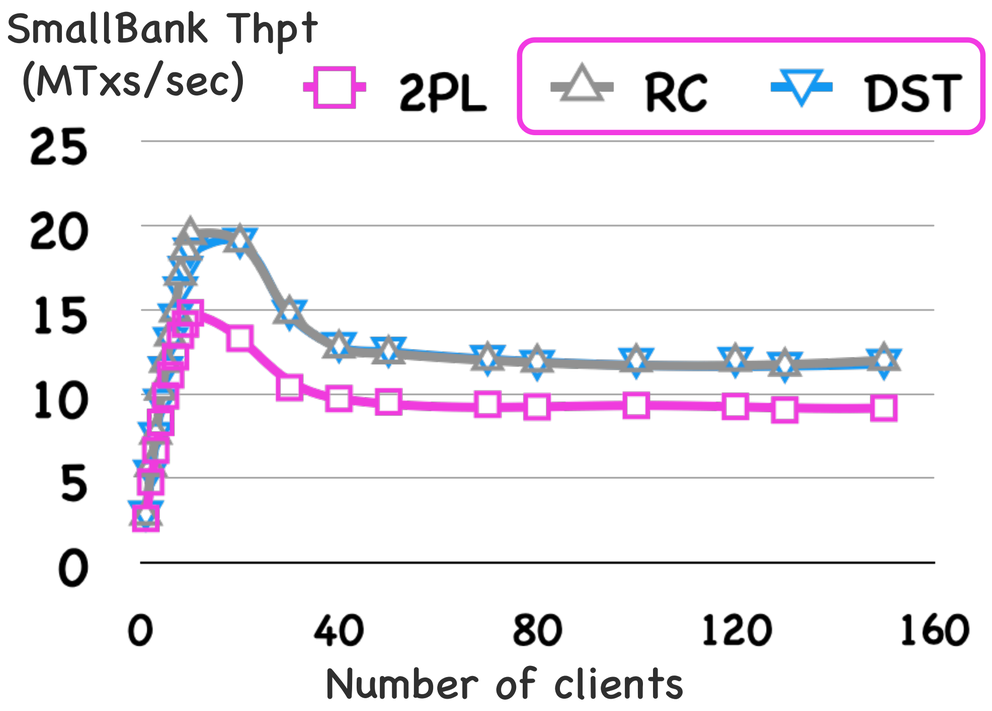
Workloads: TPC-C,Smallbank

- **DST ~= RC in performance**

Beneficial to write-mostly workloads

- **Reduce lock between read/write**



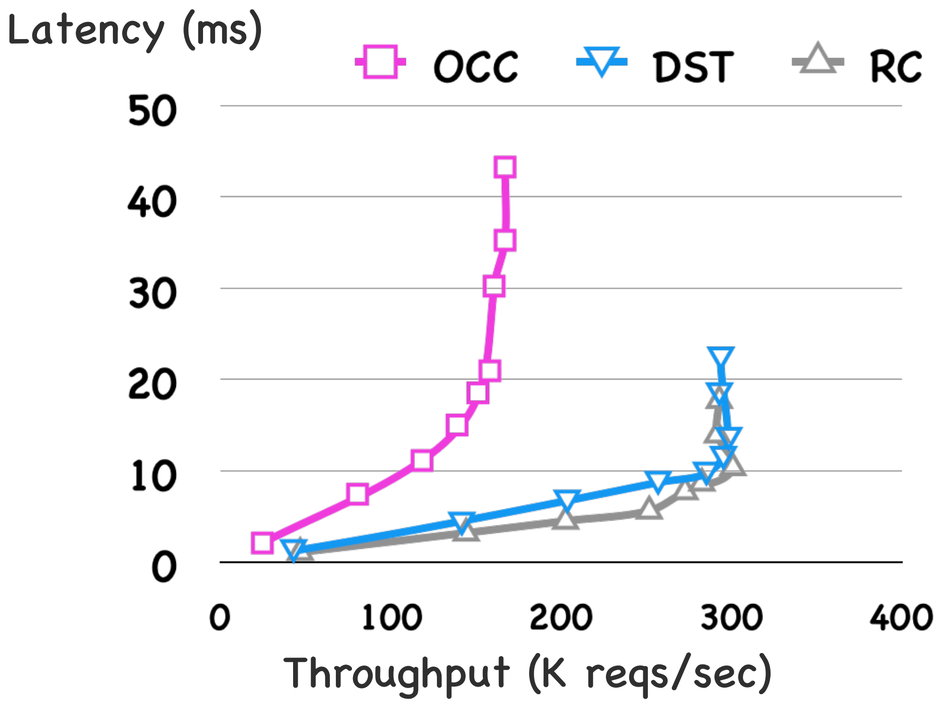


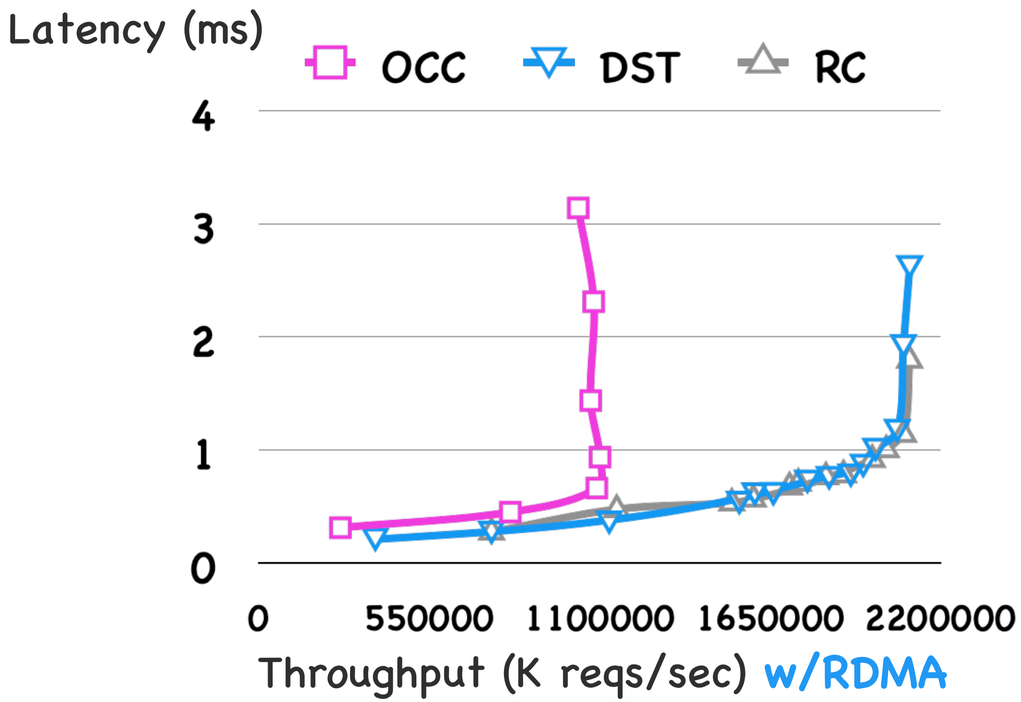
OCC X DST

Cluster Name	CPU	Network	Num
AWS	r4.2xlarge (8 vcpu)	10GbE	32
VALR	2 X 12 cores	100Gbp IB (RDMA)	16

Workloads: TPC-E (read-mostly)

- **DST also ~= RC in performance**



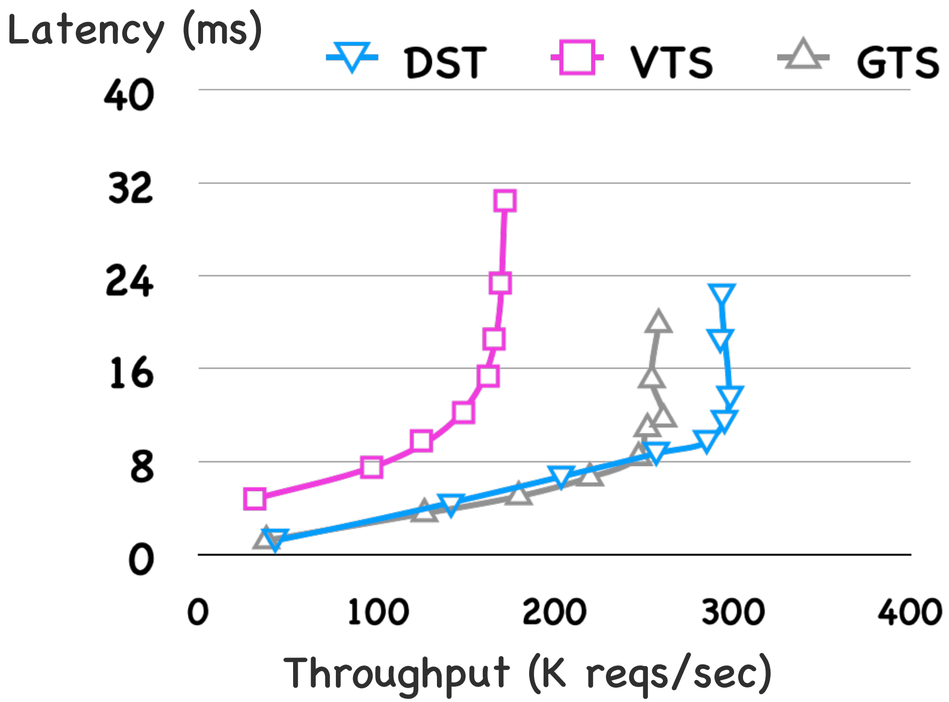


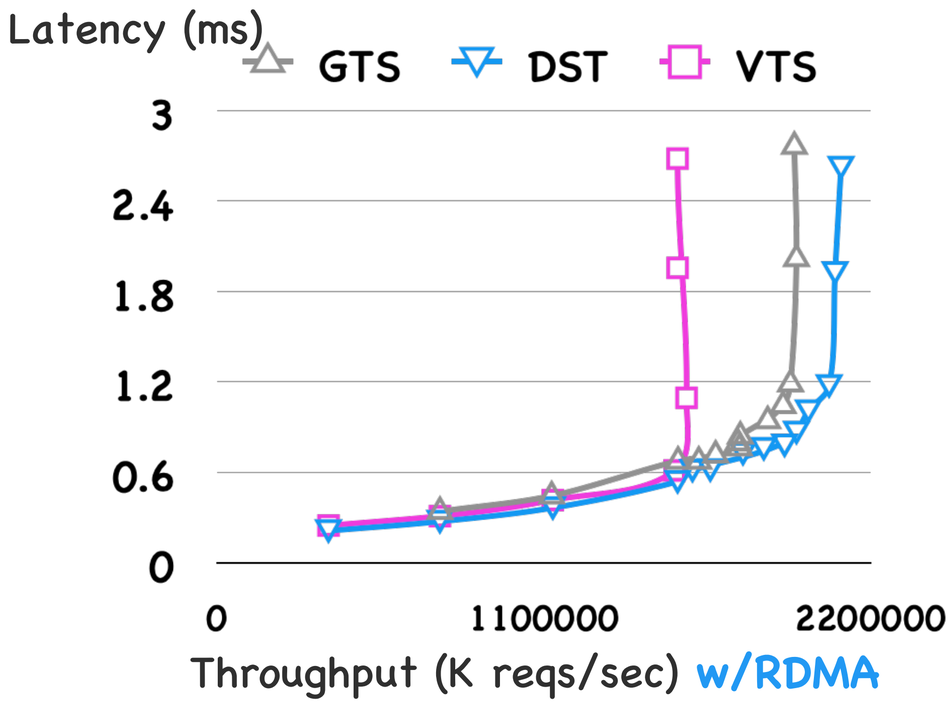
OCC X DST

Cluster Name	CPU	Network	Num
AWS	r4.2xlarge (8 vcpu)	10GbE	32
VALR	2 X 12 cores	100Gbp IB **(RDMA)**	16

Workloads: TPC-E (read-mostly)

- **Near zero-cost timestamp overhead**



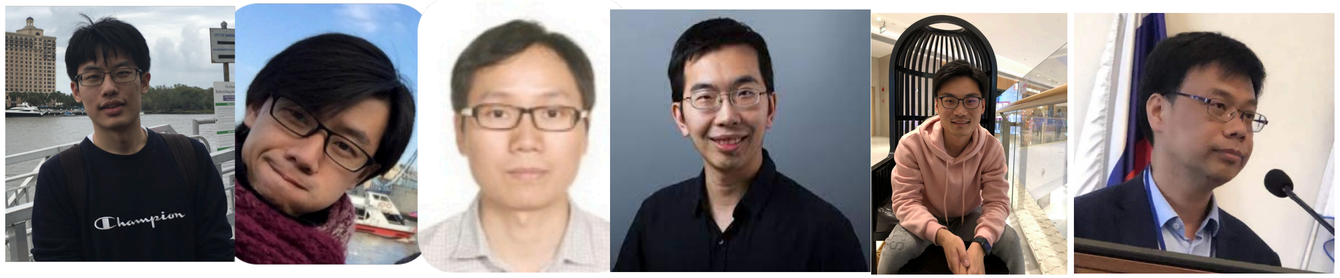


Conclusion

DST provides **serializable & fresh MVCC** in a networked setting

- Also generalized to different CCs

Efficient & Scalable: Near zero-cost timestamp overhead



IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

Thanks & QA!

Backup