Distributed Transactions

Arvind Krishnamurthy

University of Washington

Spanner

- Key features:
  - general-purpose transactions across sharded datasets
  - high performance
  - “TrueTime” API and “external consistency”
  - multi-version data store

Example: Social Network

- Consider a simple schema:
  - User posts
  - Friend lists
- Looks like a database, but:
  - shard data across multiple continents
  - shard data across 1000s of machines
  - replicated data within a continent/country
- Lock-free read only transactions

Read Transactions

- Example: Generate a page of friends’ recent posts
  - Consistent view of friend list and their posts
  - Want to support:
    - remove friend X
    - post something about friend X

Spanner Transaction

- Two-phase commit layered on top of Paxos
  - Paxos provides reliability and replication
  - 2PC allows coordination of different groups responsible for different datasets
  - Layering provides non-blocking 2PC
- Uses 2-phase locking to deal with concurrency

Example

- Consider transfer between two bank accounts

Read-only transactions

- User X sequentially performs:
  - remove friend Y
  - post something about friend Y
- User Y atomically reads X’s friends list and X’s posts
  - Display X’s posts only if X’s friends list includes Y
- Let us consider optimizing this with synchronized clocks

Synchronized Clocks

- Use multi-version data
- All updates tagged with the time of update
- Reads performed at a particular point in time
  - Called snapshot reads
  - Applications might be willing to read snapshots at some recent time in the past
- How can we make this work with partially synchronized clocks?

TrueTime

API that exposes real time, with uncertainty

`{earliest: e, latest: l} = TT.now()`

"Real time" is between `earliest` and `latest`

Time is an illusion!

If I call TT.now() on two nodes simultaneously, intervals *guaranteed* to overlap!

If intervals don't overlap, the later one happened later!

Using TrueTime

- Consider a simple write operation on a single node
- Suppose you want to associate a “write timestamp” for the operation
  - Need to ensure that the write timestamp falls during the physical time interval of the client perceived delay
- What timestamp should I attribute to the operation?
  - What should the server do to guarantee linearizability?

Using TrueTime

- When server receives write operation op:
  - set op.tstamp = TT.now().latest
  - Wait till TT.now().earliest > op.tstamp
  - Perform write: record a new version with op.tstamp
  - Send response to the client
- When server receives a “read snapshot at t” operation
  - Ensure that t < TT.earliest()
  - Read versions of objects associated with time t

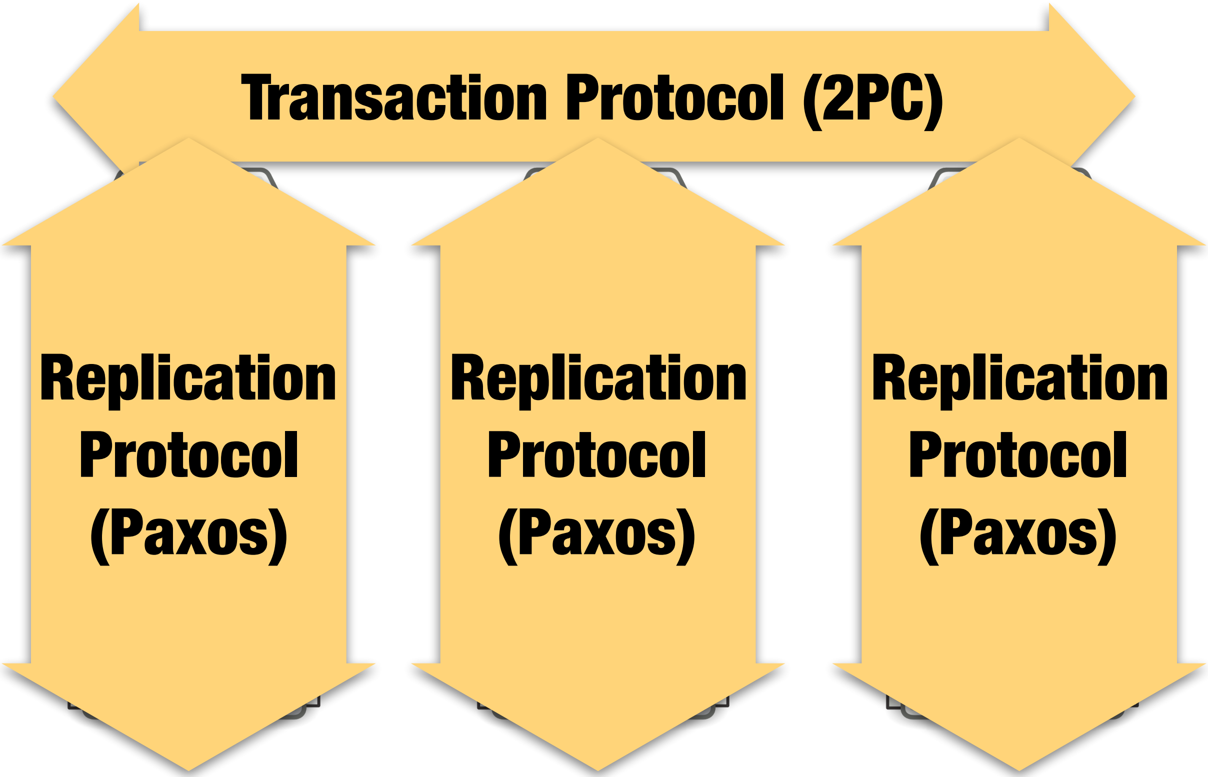
Generalizing to Transactions

- Multiple groups involved in each transaction (2PC)
- Multiple nodes involved in each group (Paxos)
- Some of the operations can be performed on the leader
    - Ensure that timestamps are monotonic across leader changes
    - Ensure that locks obtained only at leaders are sufficient

Many Protocol Details

- Sections 4.1 & 4.2:
  - Each replica determines whether its state is sufficiently up-to-date to satisfy a read
  - replica can satisfy a read at t if t <= tsafe
    - tsafe = min($tsafe^{Paxos}$, $tsafe^{TM}$)
    - $tsafe^{Paxos}$ is timestamp of last “Paxos write”
    - $tsafe^{TM}$ is timestamp of last prepare (also Paxos write)
  - Read-only transaction first identifies a timestamp and then performs a snapshot read at the timestamp
    - Timestamp can be TT.now().latest
    - Or smaller to reduce the commit wait time

Distributed transactions with strong consistency are useful



Transaction Protocol (2PC)

Replication Protocol (Paxos)

Replication Protocol (Paxos)

Replication Protocol (Paxos)

👎 **high latency**

👎 **low throughput**

TAPIR Insights

Strong replication protocols ***waste work***.

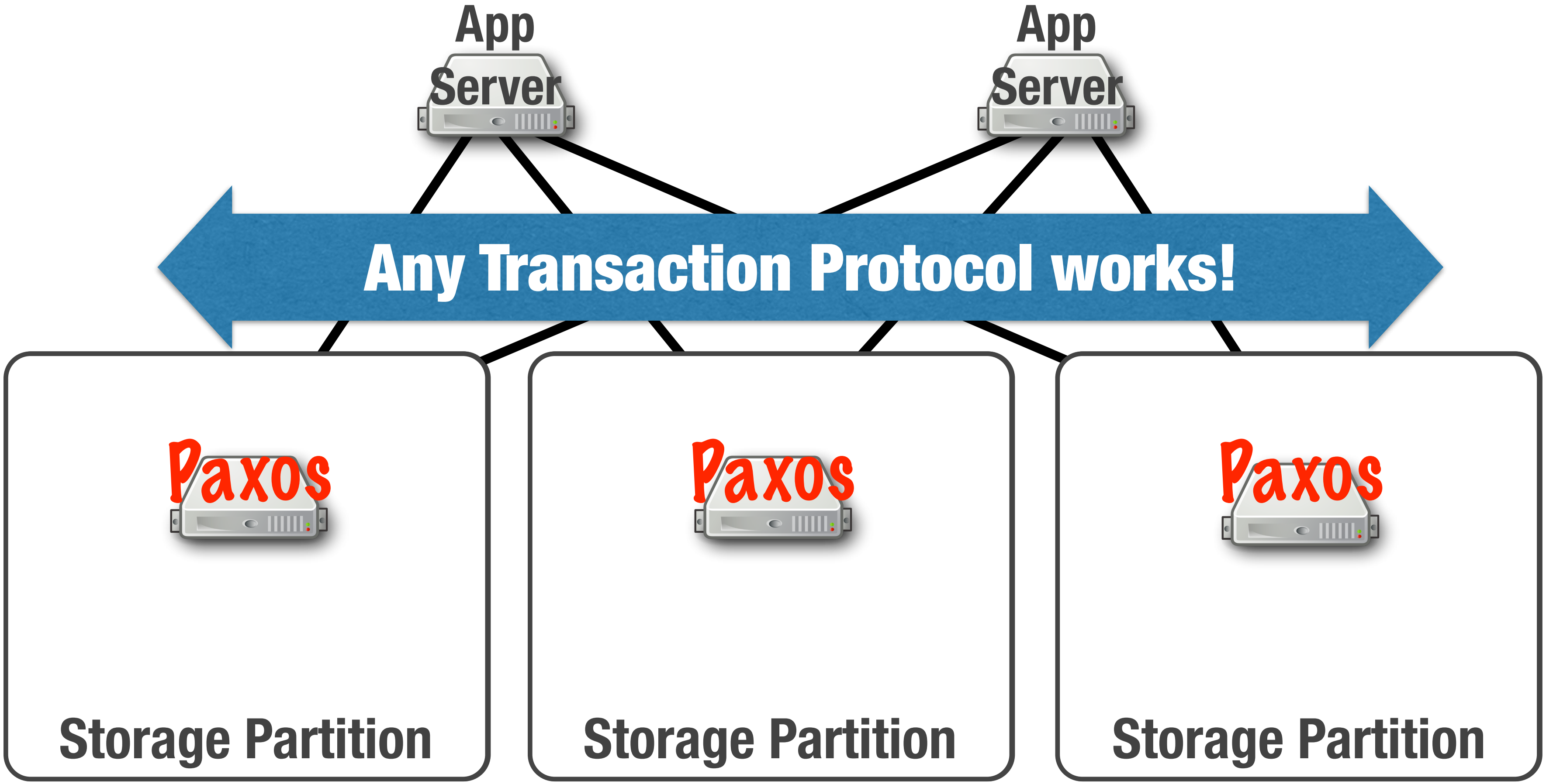
Co-design a ***linearizable*** transaction protocol with ***unordered*** replication.

Result:
cheaper transactions***,***
same ***strong*** guarantees

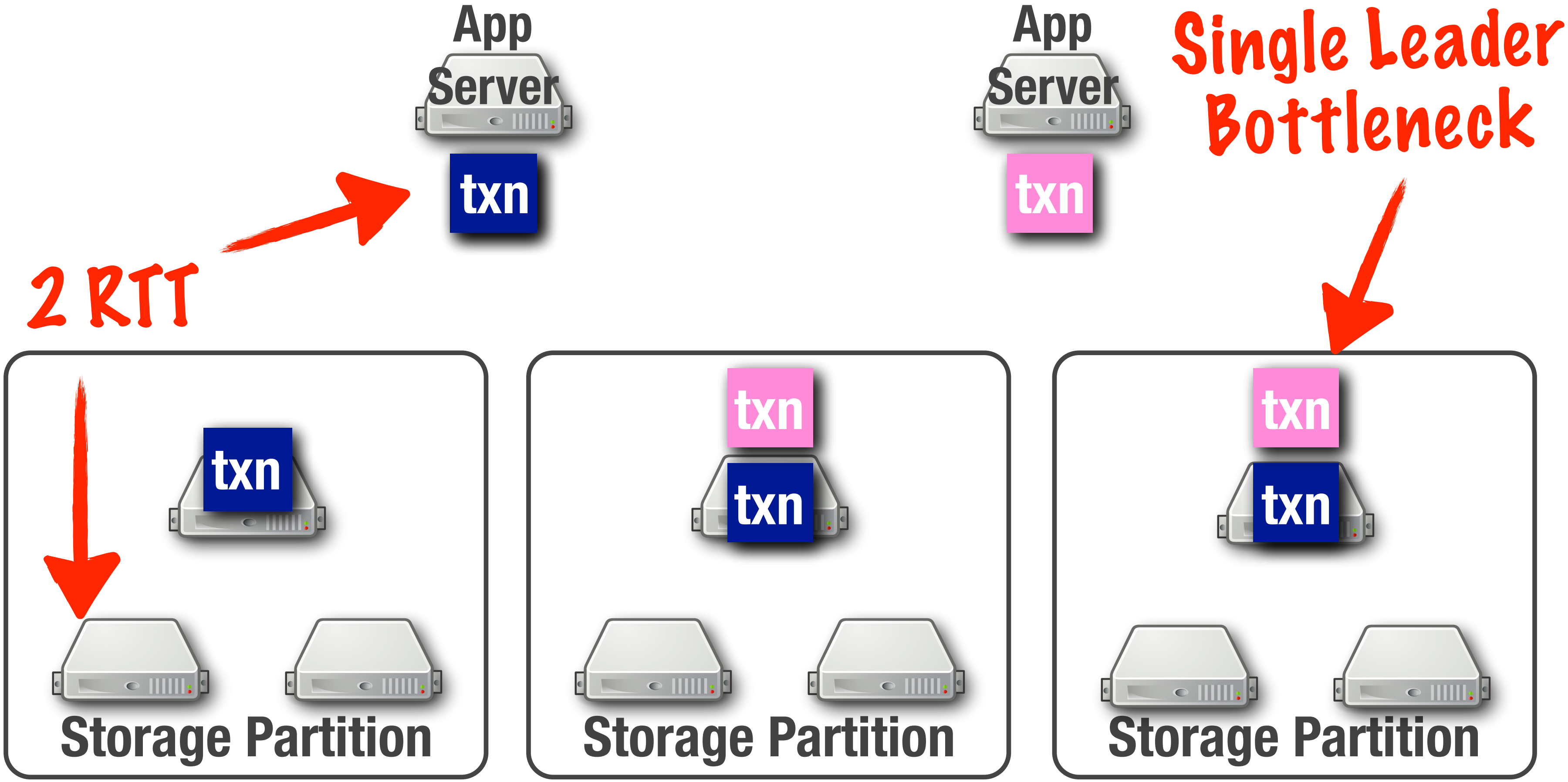
Existing transaction systems combine protocols with strong guarantees

Guarantees	Fault-tolerance	Scalability	Linearizable Ordering
Distributed Transaction Protocol		✓	✓
Replication Protocol	✓		✓

Strong replication works with existing transaction protocols



… but is expensive.



Can we reduce the cost?

Can we still ensure this?

Guarantees	Fault-tolerance	Scalability	Linearizable Ordering
Distributed Transaction Protocol		✓	✓
Replication Protocol	✓		✗

What do we need here instead?

Will it be cheaper?

Inconsistent Replication

New replication protocol providing ***unordered operations*** where replicas ***agree on operation results***.

IR Guarantees

IR provides a fault-tolerant, unordered ***operation set*** with the following guarantees:

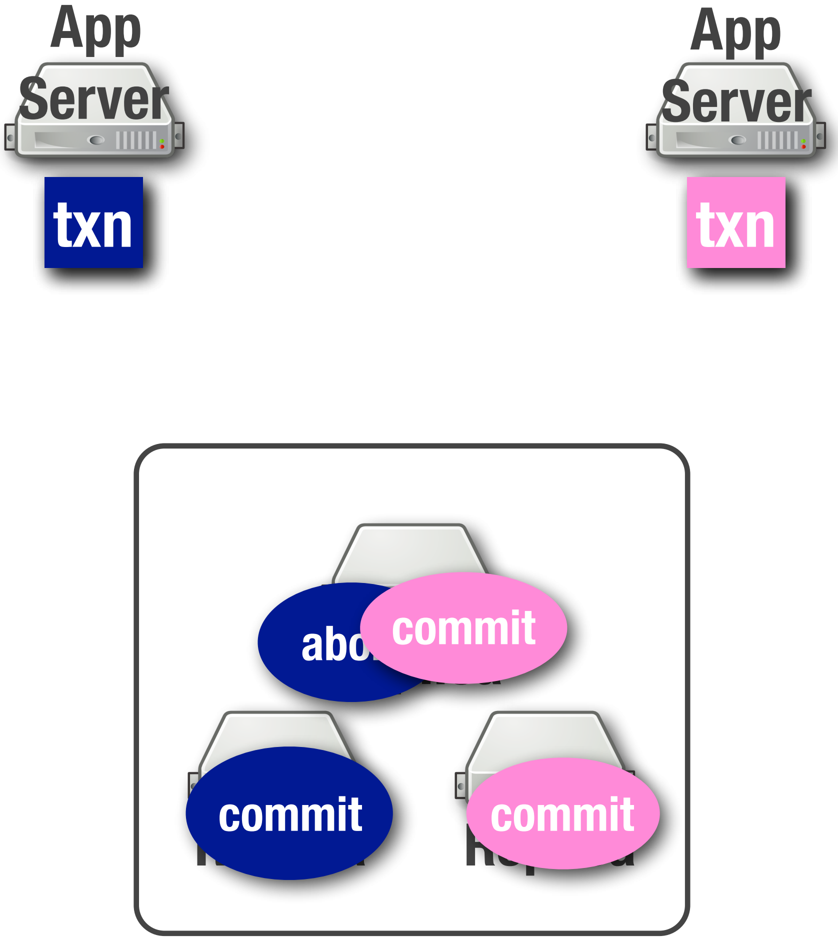
Fault-tolerance for operations and their results with up to *f* failures out of *2f+1* replicas.

Agreement from at least a majority of the replicas for any operation result.

Overlap with every previously added operation on at least one replica out of every quorum.

IR provides a way to avoid conflicts ***without*** *strong operation ordering*

- IR ensures a majority agree to every operation result.
- Quorum intersection ensures every conflict is detectable.
- IR ensures conflict decisions from application protocol are fault-tolerant.



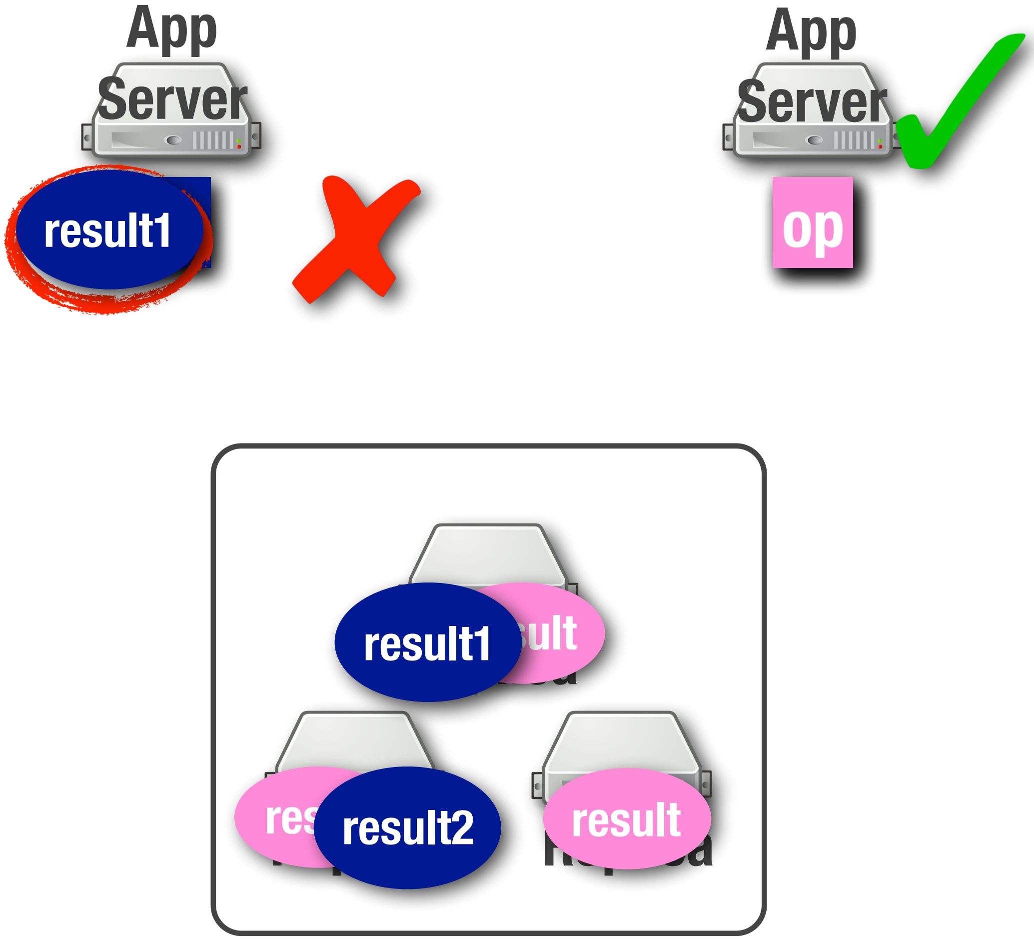
The IR Protocol (simplified)

1. Execute operation at replicas.

2. If results from a quorum match, return result.

3. If not, application protocol picks a result.

4. Update result at replicas.



IR Pros & Cons

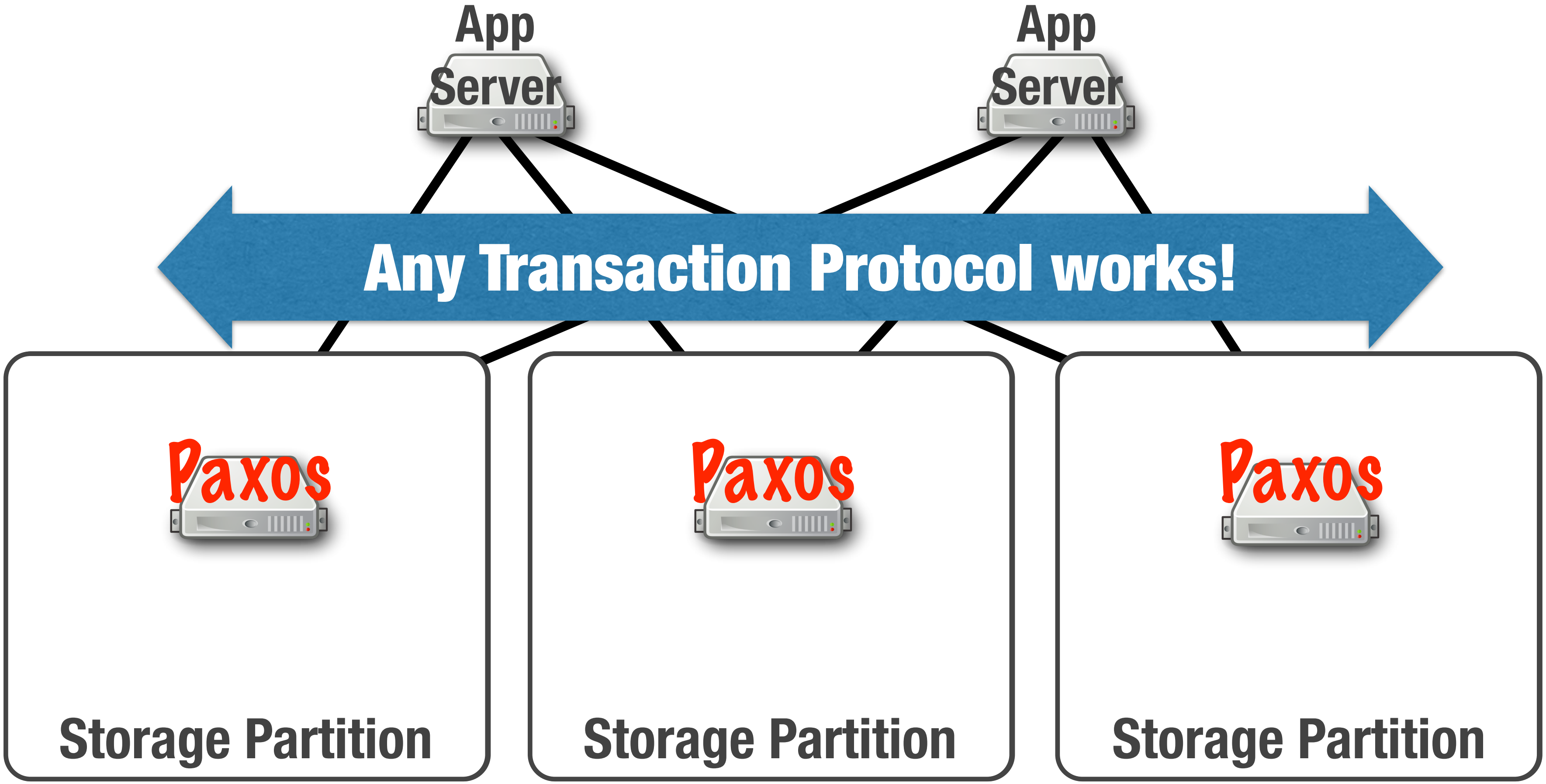
Fast: 1 round-trip fast path, 2 round-trip slow path

Efficient: No cross-replica coordination or leader needed to complete operations

Less general: Does not ensure replicas appear as a single machine

Needs co-design: Requires careful co-design for both correctness and performance

Strong replication works with existing transaction protocols

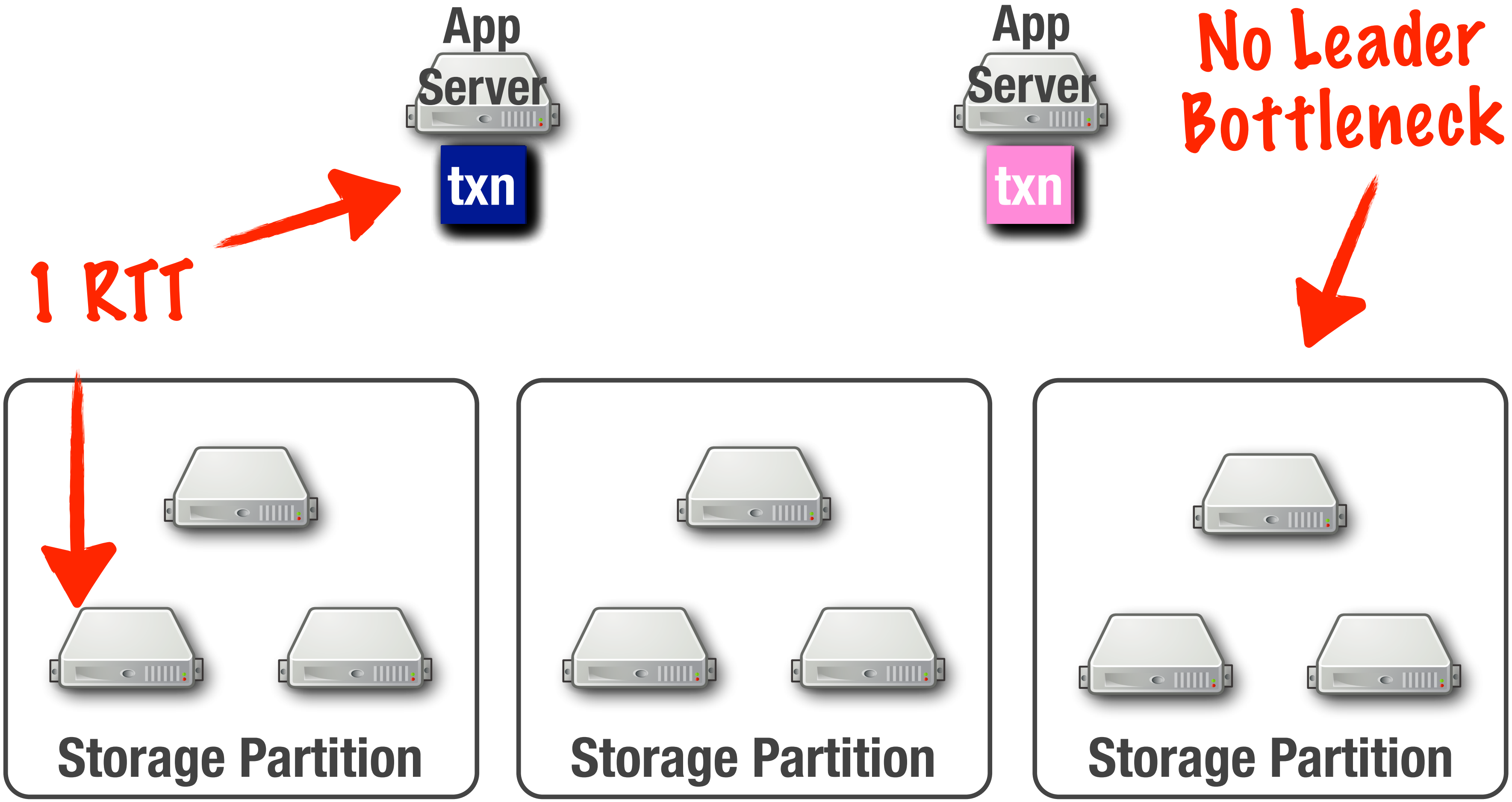


TAPIR

New distributed transaction protocol that provides linearizable transactions using IR (Inconsistent Replication).

Inconsistent Replication is unordered/unsequenced persistence of operations on replica nodes.

TAPIR coordinators are App Servers



App Server

txn

App Server

txn

No Leader Bottleneck

1 RTT

Storage Partition

Storage Partition

Storage Partition

Two key ideas

- TAPIR uses a super-quorum of nodes inside each shard to ensure recovery from failed coordinators
- TAPIR uses a form of OCC that checks for the same serialization order across different shards and nodes

Fast-Path/Slow-Path

- App Server issues “Prepares” to all nodes in all shards
- 1 RTT case: If all shards respond with Prepare-OK “super quorums”, then
  - App Server declares the transaction to be successful
  - Inform shards of the transaction commit in the background
- 2 RTT case: If all shards respond with just a Prepare-OK quorum, then
  - App Server first persists the transaction result in a coordinator shard
  - Then returns success to the application, informs shard of transaction commit

“Super Quorum” of Nodes

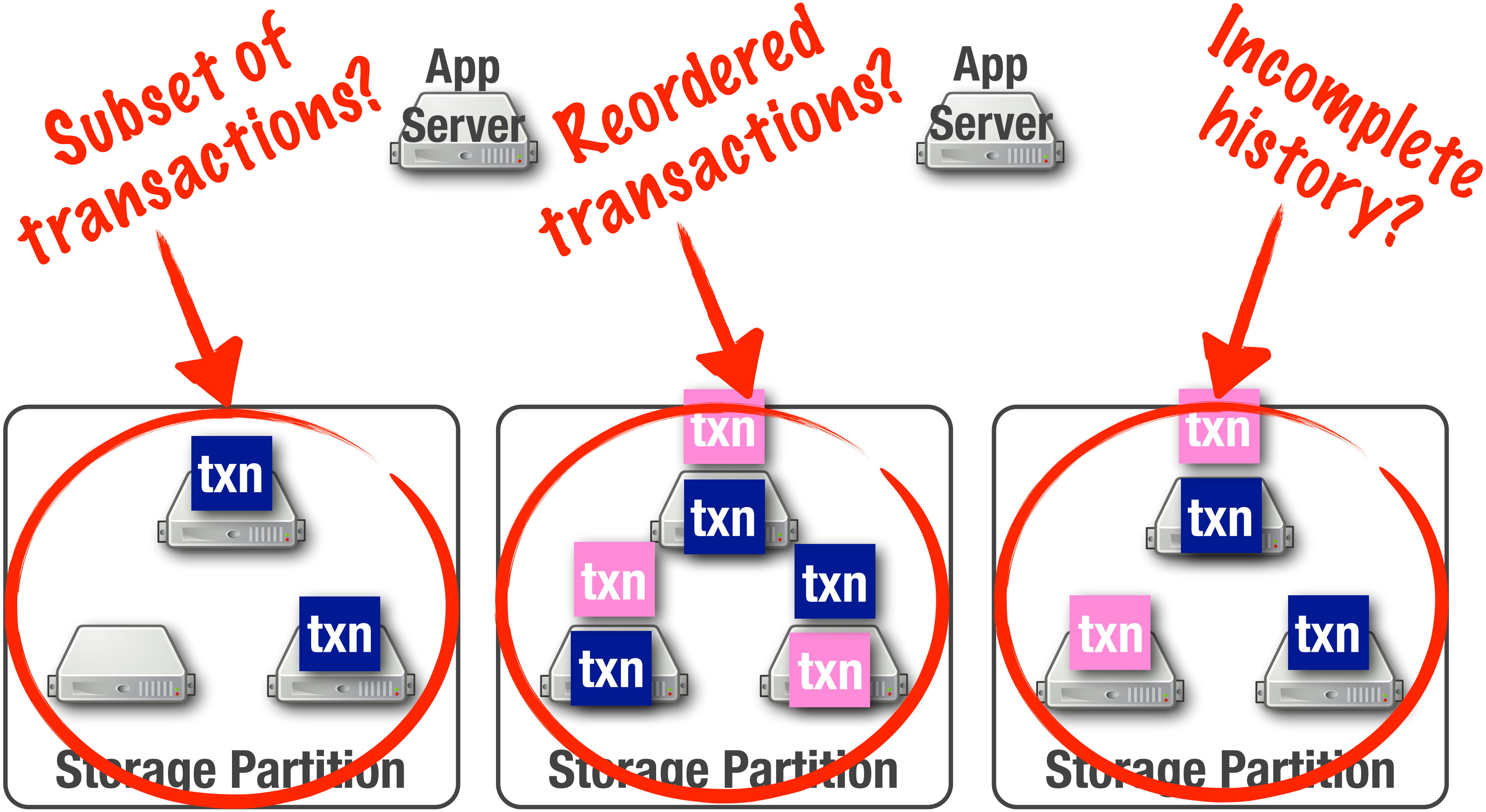
- App Server initiated operations require a “super-quorum”
  - Super quorum size is $\lceil 3f/2 \rceil + 1$
  - Recovery protocol continues only those transactions that have a majority of votes amongst live nodes
  - Recovery differentiates the following outcomes
    - Transaction committed in a fast path
    - Transaction not committed in a fast path, but serializable
    - Transaction that cannot be serialized

"Super Quorum" of Nodes

- Let us say T1, T2 are two conflicting transactions
  - T1 receives $\lceil 3f/2 \rceil + 1$ votes, T2 receives $\lfloor f/2 \rfloor$ votes
    - Even after f failures, T1 has a majority of votes
    - Recovery protocol will never attempt to commit T2
  - T1 receives $\lceil f/2 \rceil + 1$ votes, T2 receives none, f nodes fail
    - Recovery protocol will attempt to commit T1
- Invariant: any transaction committed in the fast path will be recovered

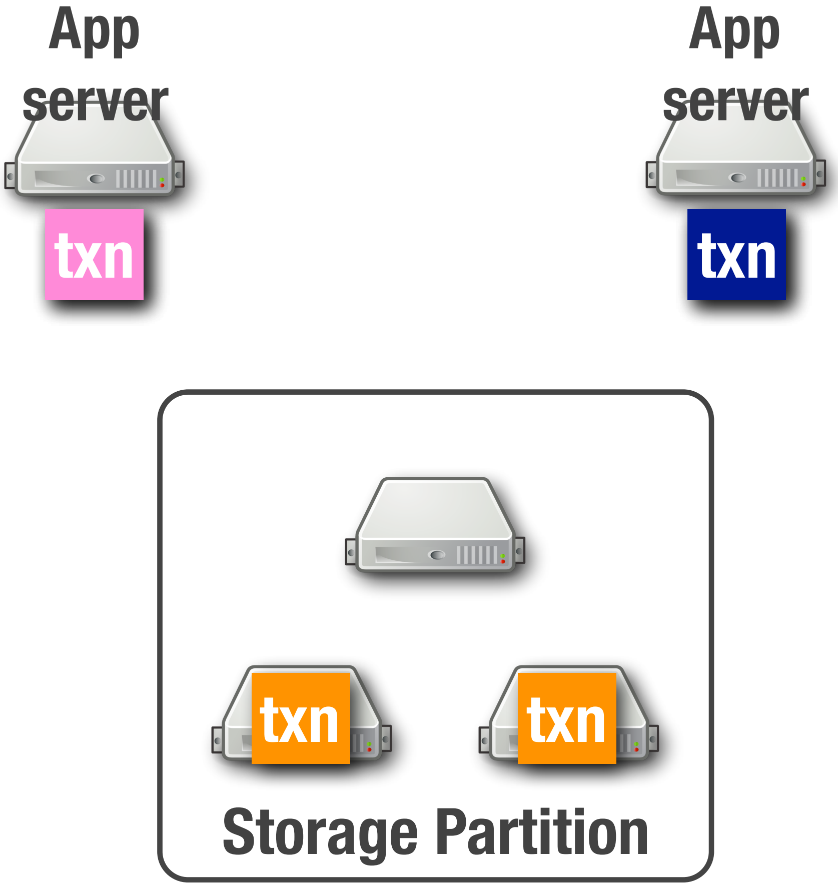
- What are the downsides of using a super quorum?

IR introduces challenges



TAPIR uses optimistic concurrency control (OCC) to detect conflicts on IR

- OCC checks one transaction at a time.
- IR ensures every pair of transactions is checked on at least one replica.
- OCC+IR ensures that every conflict is detected.



Why is constrained OCC needed?

- Consider three transactions starting with X=Y=Z=0

  T1: Read X -> 0; Y = 1

  T2: Read Y -> 0; Z = 1

  T3: Read Z -> 0; X = 1

- Shard-wise traditional OCC checks:

  X’s shard: OK (“T1 before T3”)

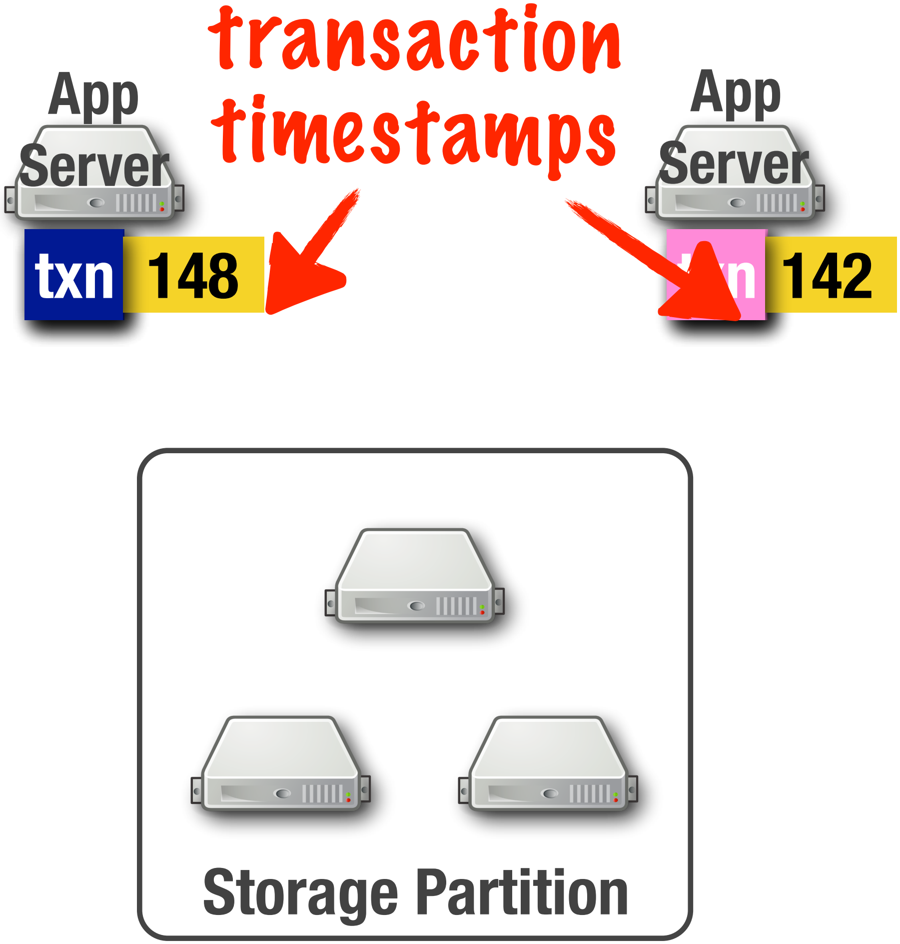
  Y’s shard: OK (“T2 before T1”)

  Z’s shard: OK (“T3 before T2”)

- Additional coordination required to see whether shard-wise OKs can yield consistent ordering of transactions

TAPIR uses loosely synchronized clocks to efficiently order transactions

- Clients pick transaction timestamp using local clock.
- Replicas validate transaction at timestamp, regardless of when they receive the transaction.
- Clock synchronization for performance, not correctness.
- Multiple outcomes: Prepare-OK, Abort, Abstain, Retry

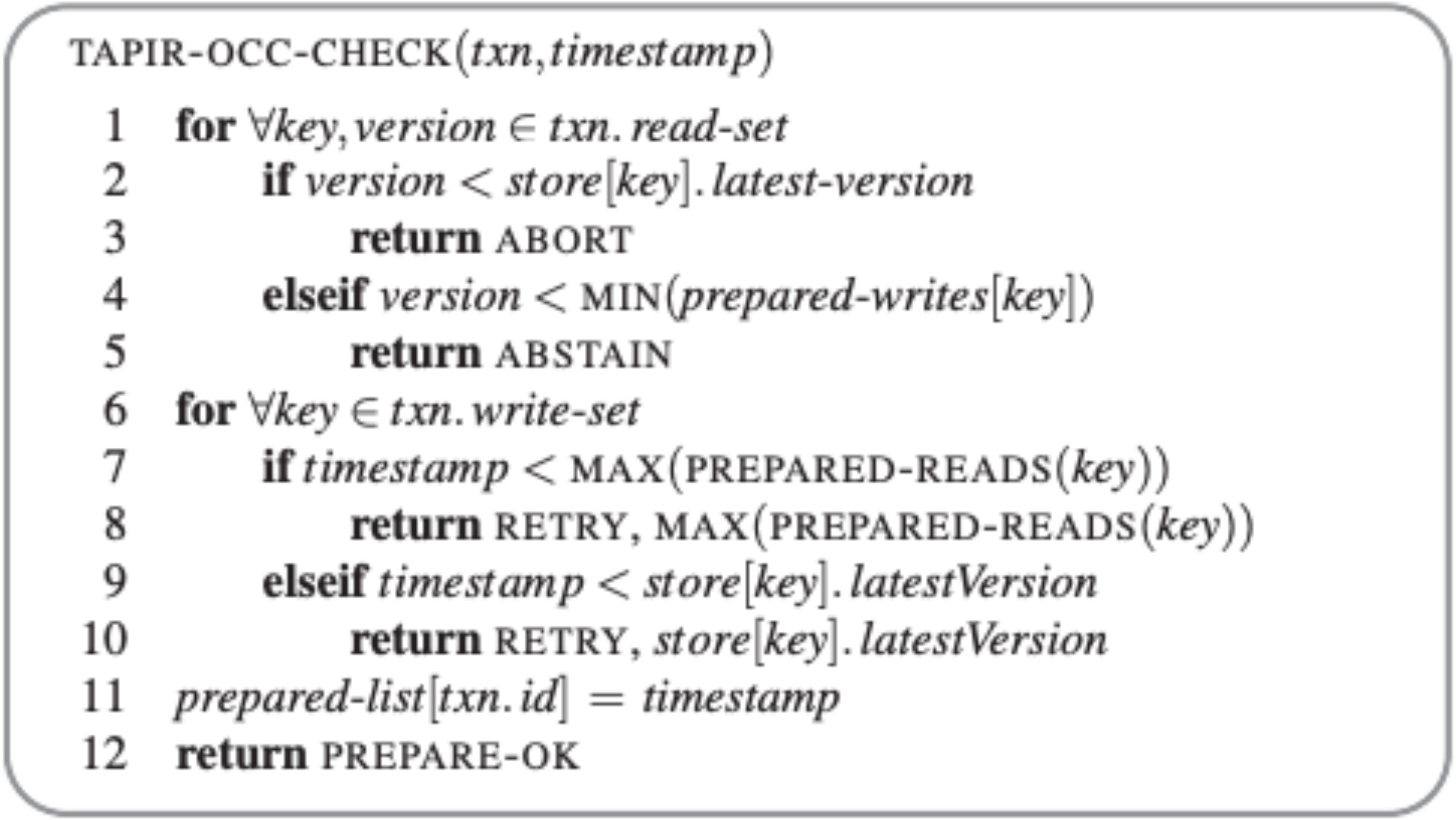


OCC Algorithm

- Consider a distributed, but non-replicated setup
- App Server requests a transaction to be serialized at time “t”
- Each server (shard) maintains:
  - Versioned memory for each key-value
  - A list of accepted transactions and a list of prepared transactions
- What should be the local OCC check?

OCC Check

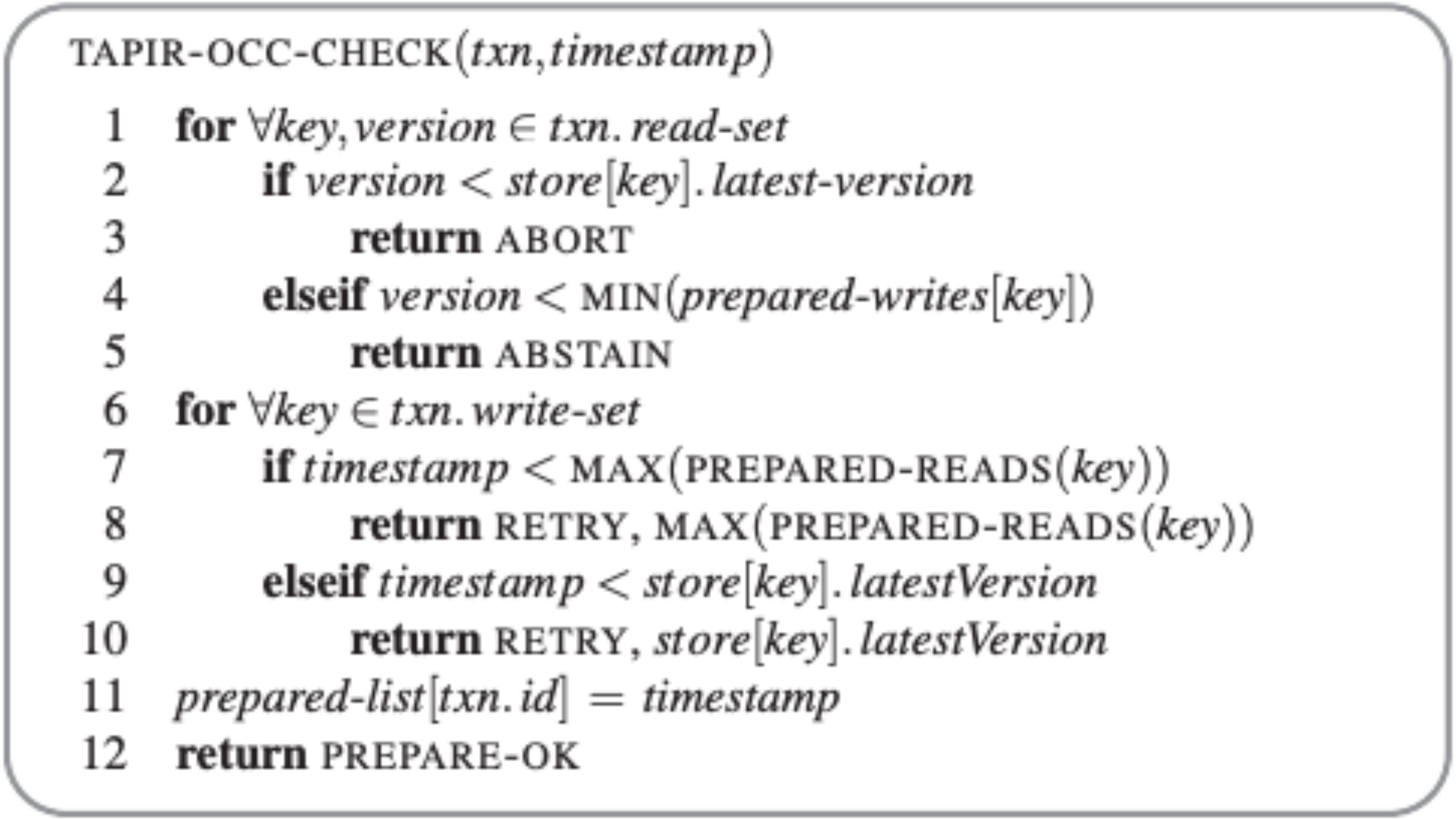
- If txn has read a key and its value has been overwritten before the timestamp, then Abort



```
TAPIR-OCC-CHECK(txn, timestamp)
for ∀key, version ∈ txn.read-set
    if version < store[key].latest-version
        return ABORT
    elseif version < MIN(prepared-writes[key])
        return ABSTAIN
for ∀key ∈ txn.write-set
    if timestamp < MAX(PREPARED-READS(key))
        return RETRY, MAX(PREPARED-READS(key))
    elseif timestamp < store[key].latestVersion
        return RETRY, store[key].latestVersion
prepared-list[txn.id] = timestamp
return PREPARE-OK
```

OCC Check

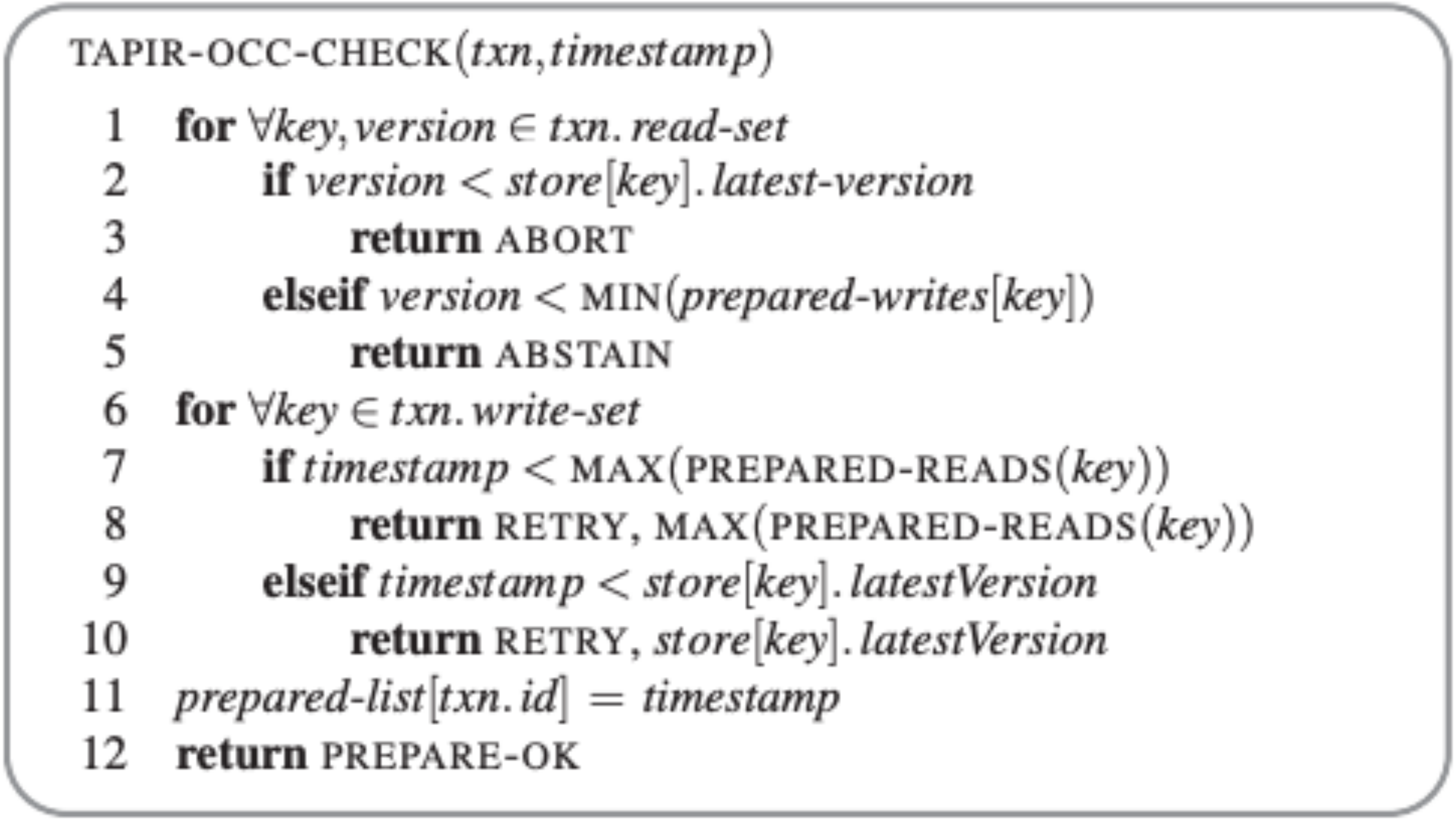
- If a prepared transaction is going to overwrite before the timestamp, then Abstain



```
TAPIR-OCC-CHECK(txn, timestamp)
for $\forall key, version \in txn.read\text{-}set$
    if $version < store[key].latest\text{-}version$
        return ABORT
    elseif $version < \text{MIN}(prepared\text{-}writes[key])$
        return ABSTAIN
for $\forall key \in txn.write\text{-}set$
    if $timestamp < \text{MAX}(\text{PREPARED-READS}(key))$
        return RETRY, $\text{MAX}(\text{PREPARED-READS}(key))$
    elseif $timestamp < store[key].latestVersion$
        return RETRY, $store[key].latestVersion$
$prepared\text{-}list[txn.id] = timestamp$
return PREPARE-OK
```

OCC Check

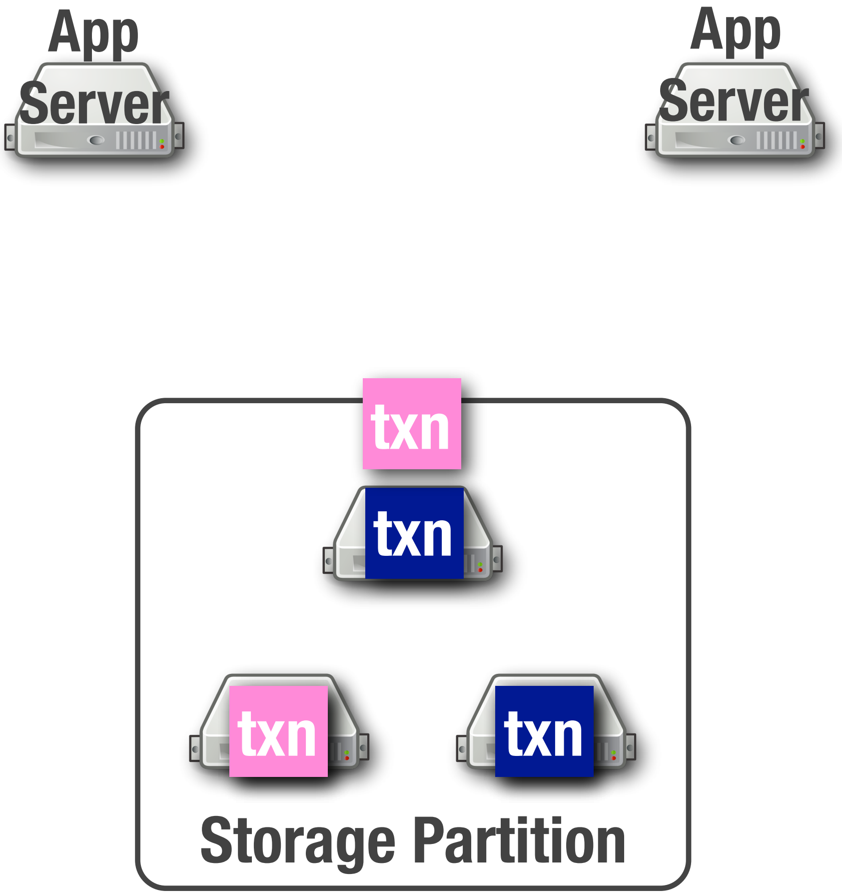
- If the key that txn attempt to write has been read by a later transaction (either prepared or committed)



```
TAPIR-OCC-CHECK($txn, timestamp$)
for $\forall key, version \in txn.read\text{-}set$
    if $version < store[key].latest\text{-}version$
        return ABORT
    elseif $version < \text{MIN}(prepared\text{-}writes[key])$
        return ABSTAIN
for $\forall key \in txn.write\text{-}set$
    if $timestamp < \text{MAX}(\text{PREPARED-READS}(key))$
        return RETRY, $\text{MAX}(\text{PREPARED-READS}(key))$
    elseif $timestamp < store[key].latestVersion$
        return RETRY, $store[key].latestVersion$
$prepared\text{-}list[txn.id] = timestamp$
return PREPARE-OK
```

TAPIR uses multi-versioning to reconcile inconsistent replicas

- IR periodically synchronizes inconsistent replicas.
- TAPIR inserts versions using the transaction timestamp.
- OCC prevents inconsistent replicas from violating transaction ordering.



Benefits of IR/TAPIR co-design

Fast: Commit transactions in 1 round-trip

Strong: Linearizable read/write transactions

Easy to use: No change in storage interface

TAPIR Measurements

How does TAPIR improve **throughput & latency**?

How does IR affect TAPIR's **abort rates**?

How does TAPIR/IR compare to **weak consistency** (e.g., Redis transactions)?

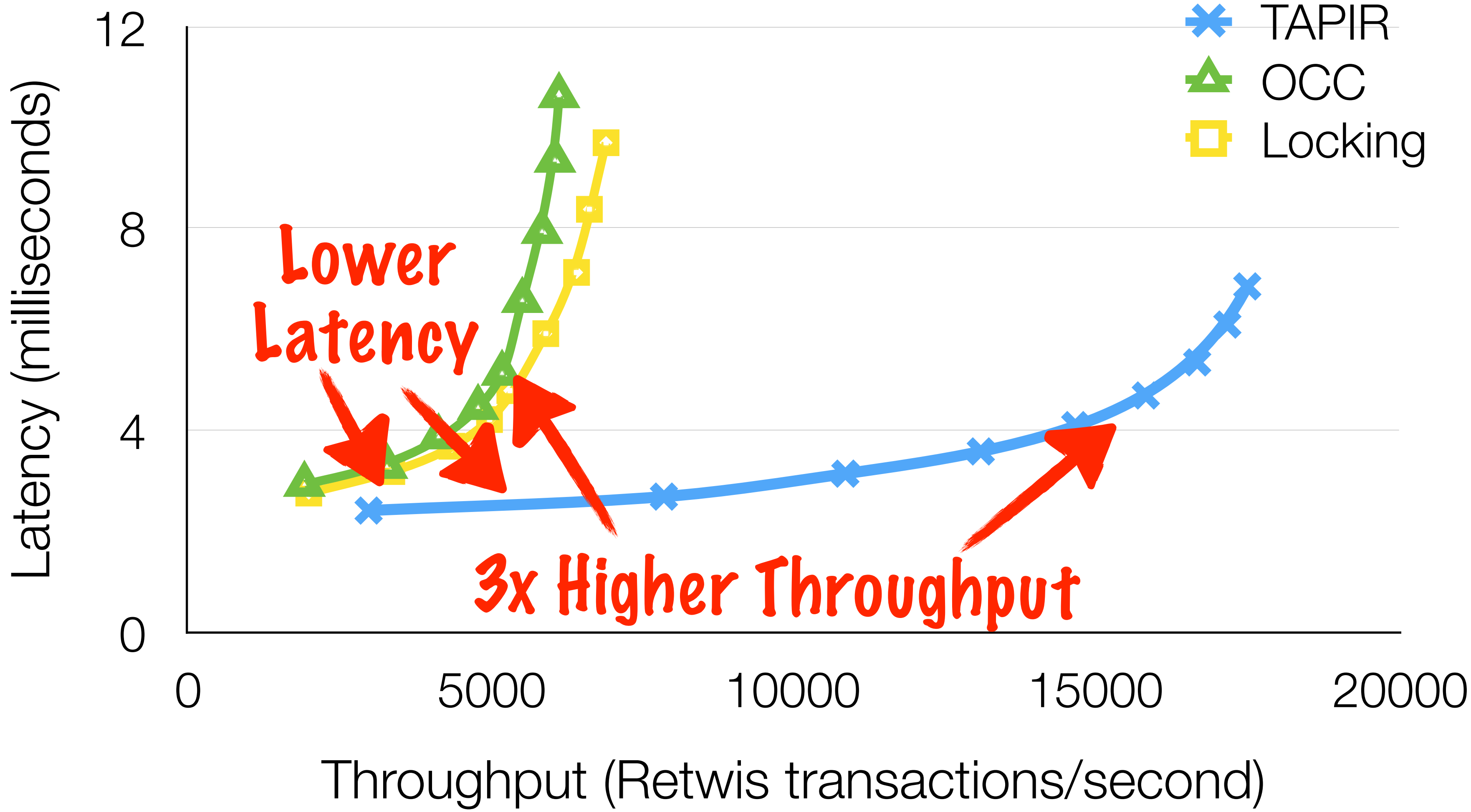
Experimental Setup

Implementation: Transactional key-value store

Workloads: Retwis Twitter clone & YCSB-t.

Testbed: Google Compute Engine VMs with ***default*** clock synchronization.

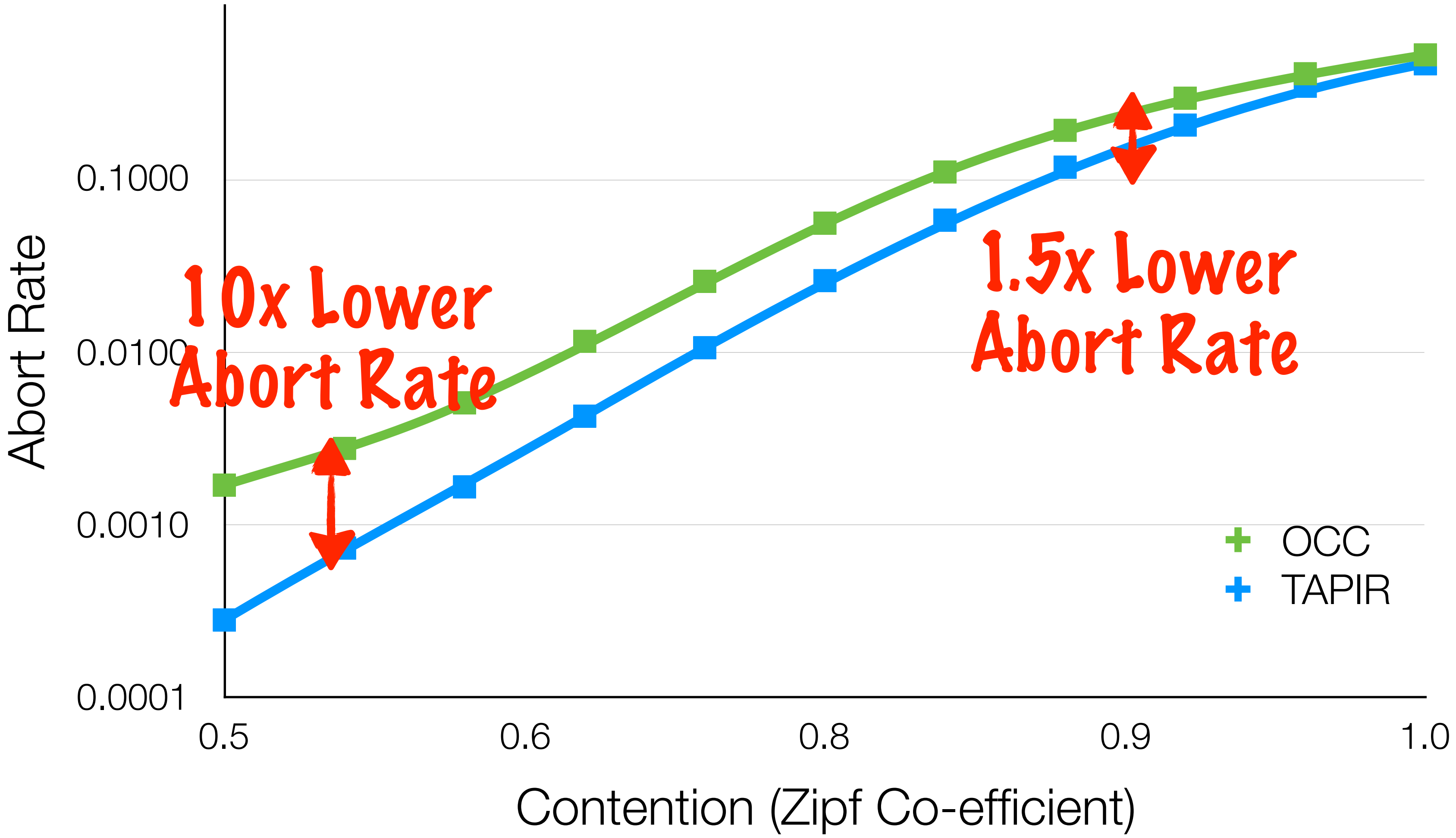
Does TAPIR & IR improve performance compared to conventional protocols?



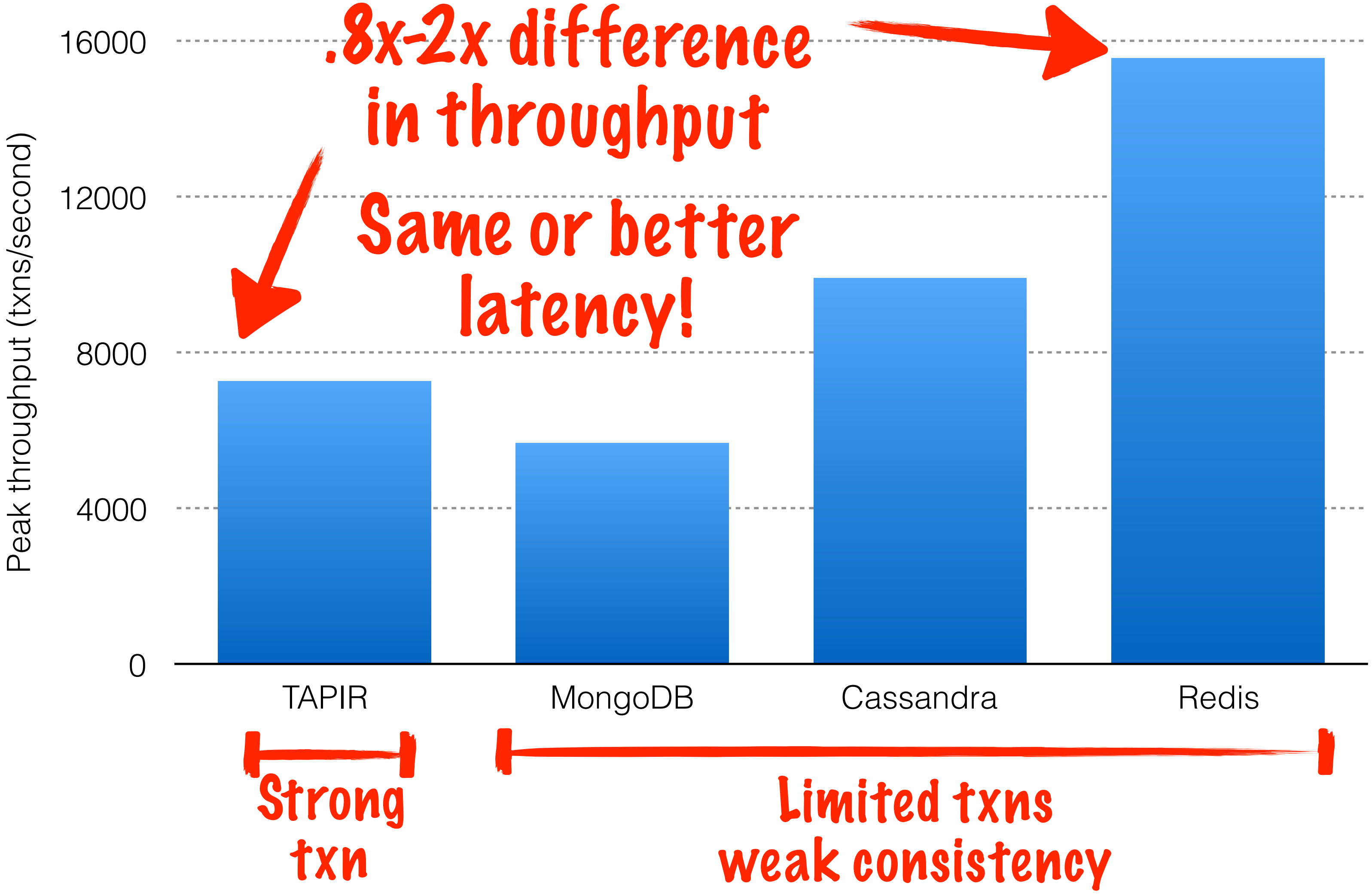
Lower Latency

3x Higher Throughput

Does IR hurt TAPIR's abort rate?



Can TAPIR & IR compete with weak consistency storage systems?



High Contention

While it is difficult for most protocols to handle high contention, TAPIR's performance is likely to degrade less gracefully than a locking-based protocol.

Summary

- Existing transactional storage systems waste work using strong replication.
- Co-design TAPIR & IR to provide linearizable transactions using an unordered replication.
- TAPIR & IR improves commit latency by 2x and throughput by 3x from conventional protocols.