The L²AW Theorem

Local Reads and Linearizable Asynchronous Replication

VLDB '25

Research Track — Distributed Transactions II

A. Katsarakis[*,†], E. Giortamis[*,♣], V. Gavrielatos[†], P. Bhatotia[♣], A. Dragojevic[♦],
B. Grot[♠], V. Nagarajan[♠], P. Fatourou[♥]

[†] Huawei Research, [♣] TU Munich, [♦] OpenAI, [♠] University of Edinburgh, [♥] University of Crete and FORTH,
[*] Equal contribution

Distributed datastores

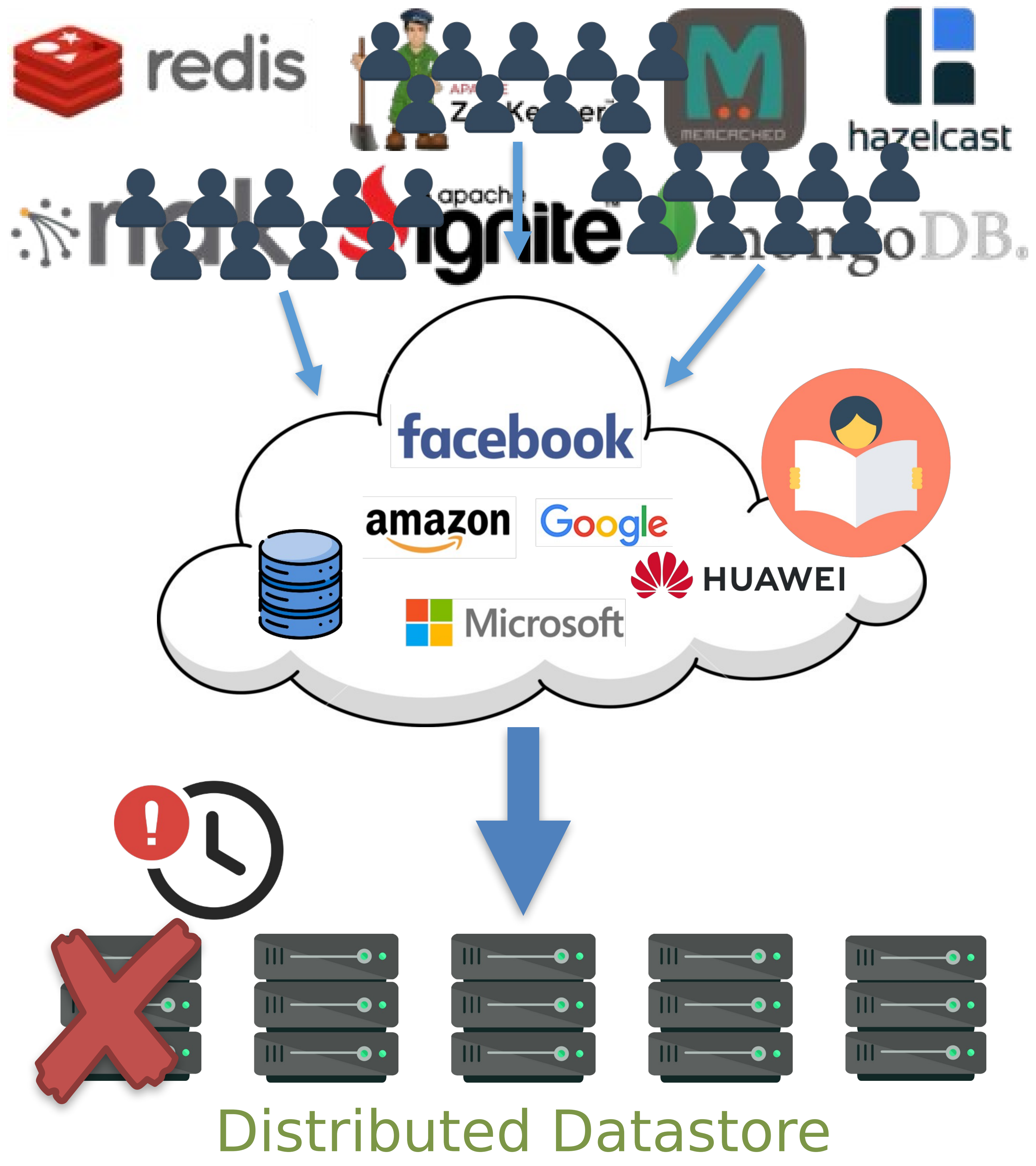
Such as KV Stores, Caches, Coordination services keep data in-memory and expose a read/write API.

Backbone of online services and DBMSes.

Characterized by

- Numerous **concurrent requests.**
- **Read-intensive** workloads.
- Need for **reliability**

  run on **fault-prone HW with unpredictable delays.**



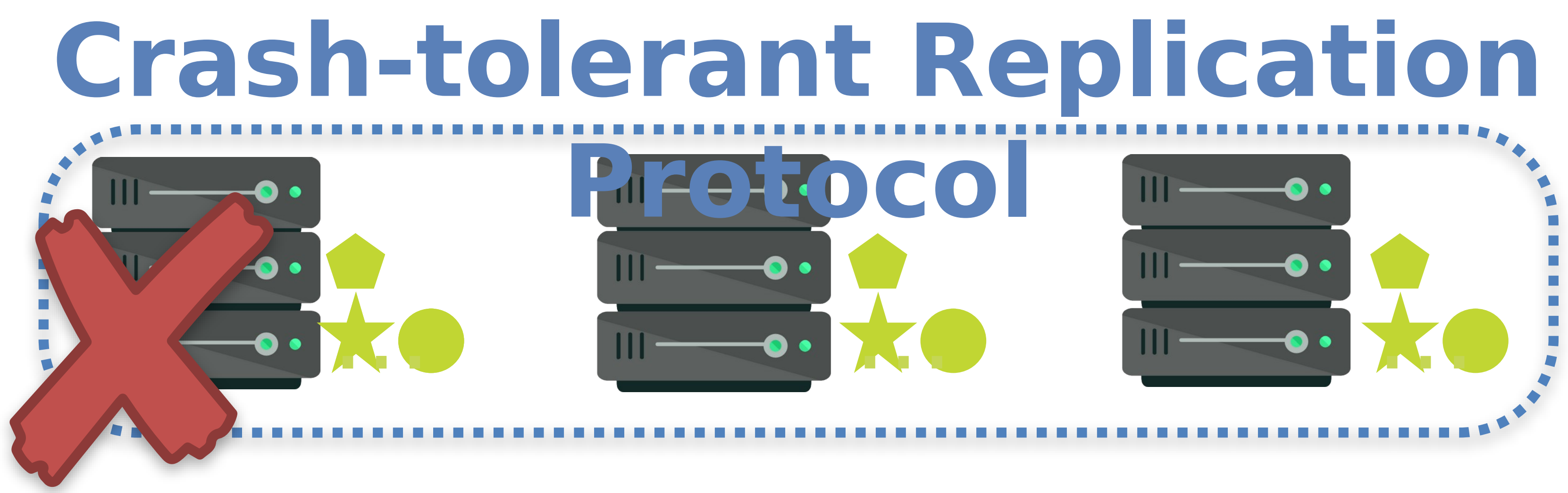
What should the ideal datastore look like?

Ideal Reliable Datastore: Performant, Consistent, & Asynchronous

Should offer

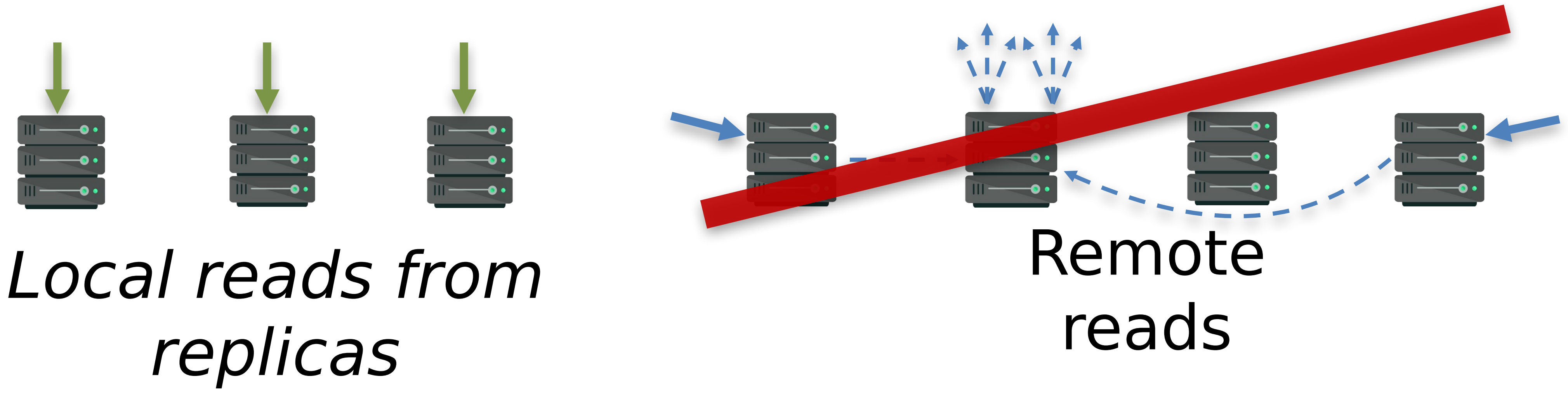
- **Crash-tolerance**: via replicated data
- **High performance**: especially for reads
- **Strong consistency** under **asynchrony**

  safe — even when timeouts do not hold.



Crash-tolerant Replication Protocols

Tolerate crashes without blocking
by determining actions to execute *reads* and *writes.*



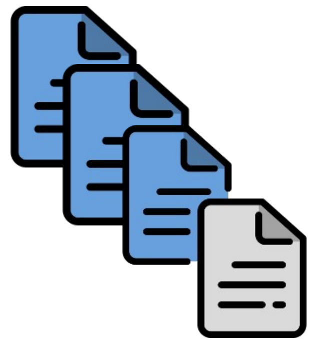
Ideal protocol features

1. **Local Reads**

   reads complete on a single replica—no inter-replica exchanges.
2. **Linearizability**

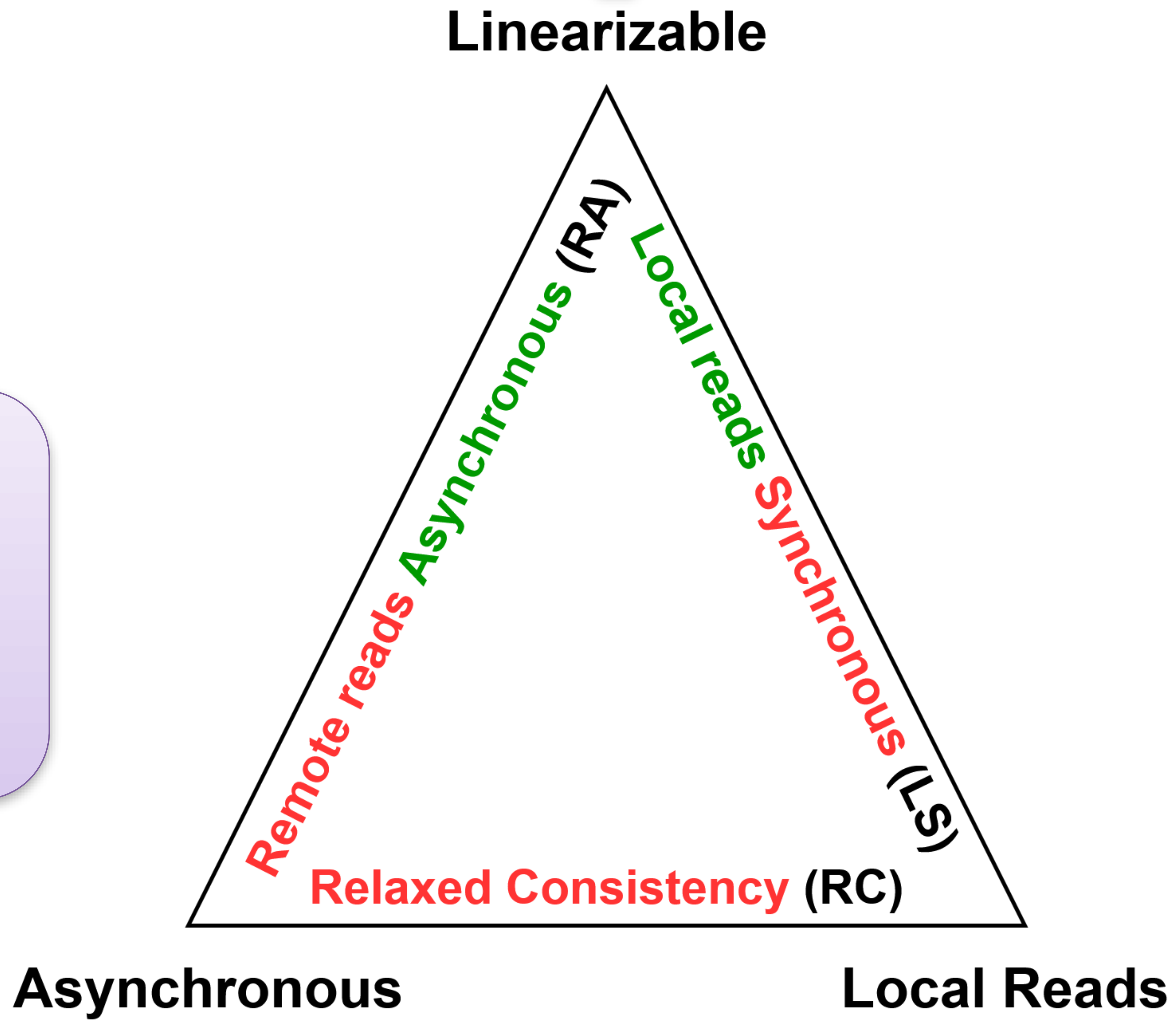
   reads/writes appear to occur instantaneously—as if a single copy.
3. **Asynchrony**

   correct in real-world deployments, even with unpredictable delays.



Do state-of-the-art protocols offer the 3 desired features?

Crash-tolerant protocols: State-of-the-art offers 2 of 3!



RA protocols: e.g. Raft, Paxos, ABD
- ✗ Remote (costly) reads
- ✓ Asynchronous
- ✓ Linearizable

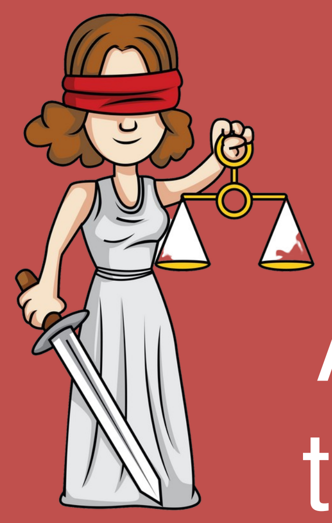
LS protocols: e.g. Hermes, CRAQ
- ✓ Local reads
- ✗ Synchronous
- ✓ Linearizable

RC protocols: e.g. ZAB
- ✓ Local reads
- ✓ Asynchronous
- ✗ Relaxed Consistency

Existing protocols: up to 2 of 3! Hints at a fundamental trade-off…

… which we prove!

Introducing the **L²AW Theorem:**



<u>The L²AW Theorem</u>

Any **L**inearizable **A**synchronous read/write register implementation that tolerates a crash (**W**ithout blocking reads or writes), has no **L**ocal reads!

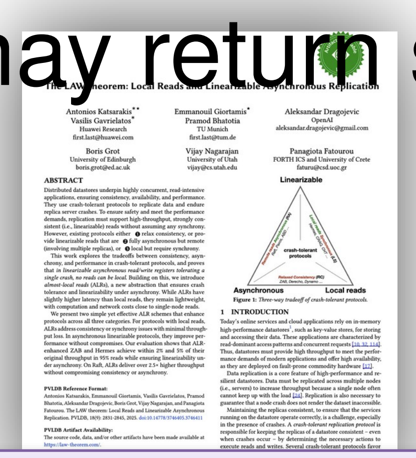
Known facts:

1. For linearizability, every read must return the most recently completed write.
2. Local reads, which avoid *coordination* (i.e., accessing other replicas), can satisfy this if no crashes occur.
3. Under asynchrony, a slow replica is indistinguishable from a crashed one.

Without coordination, a replica cannot know if it is considered crashed by others its local read may return stale data.

Intuition of the proof: 

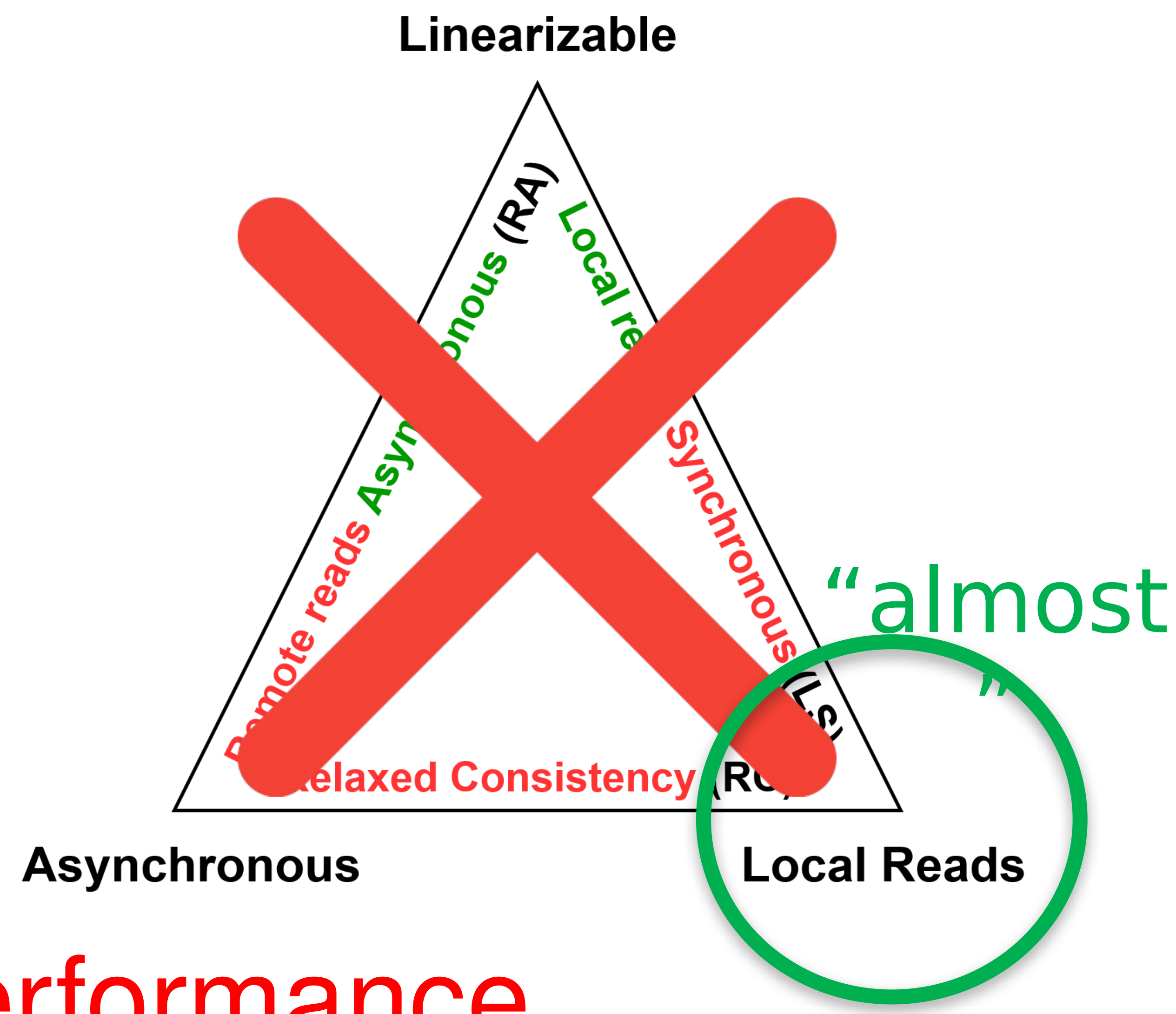
We prove (via indistinguishability) that **all crash-tolerant asynchronous implementations that are linearizable cannot allow even a single local read.**



Formal proof and more details in our

Does that mean we cannot improve on state-of-the-art?

If local reads are impossible under asynchrony & linearizability…
Can we improve read performance without breaking L^2AW?



Case study: RA protocols (e.g., Paxos, Raft)

Problem: expensive coordination for *every* read request ☾ poor performance

Insight

- The **L^2AW affects latency** but **not necessarily throughput** of reads.
- Asynchronous linearizable reads need not be as costly as in RA protocols

Can we make reads “almost local” under asynchrony—
cheap, safe, and applicable across protocols?

Enter *Almost-local Reads* (ALRs)

Almost Local Reads (ALRs)

a **batch-based abstraction with a twist**

Key Idea

- **One lightweight remote *sync* operation**
  to complete a **batch** of otherwise locally executed **reads**.
- The cost of sync is **independent of the batch** size and contents!

Implications

- Remote access to other replicas still there ☾ inevitable (L^2AW result!)
- But very low cost — computation & network cost close to local reads
  ☾ no network messages or CPU cycles for each individual read

example of reads invoked by a replica	with ALRs
$read_1(x)$	local
$read_2(y)$	local
…	…
$read_n(z)$	local
	sync

ALR batch

The **only** remote op. ☾ low constant cost

Unlike typical batching, ALR batches = low constant cost to remote replicas !

ALRs for RA protocols

<u>Background</u>: most RA protocols follow the *state machine replication* model

- reads and writes are serialized and applied in the same order to all replicas
- reads, thus, require remote coordination via majority quorums

<u>How ALRs work:</u>

a) Form a batch with the pending reads and associate a *sync* with it.

b) Buffer (but not execute) the reads while the sync is in flight

- the sync acts like a ”fake” write that does not alter state but executes the write algorithm

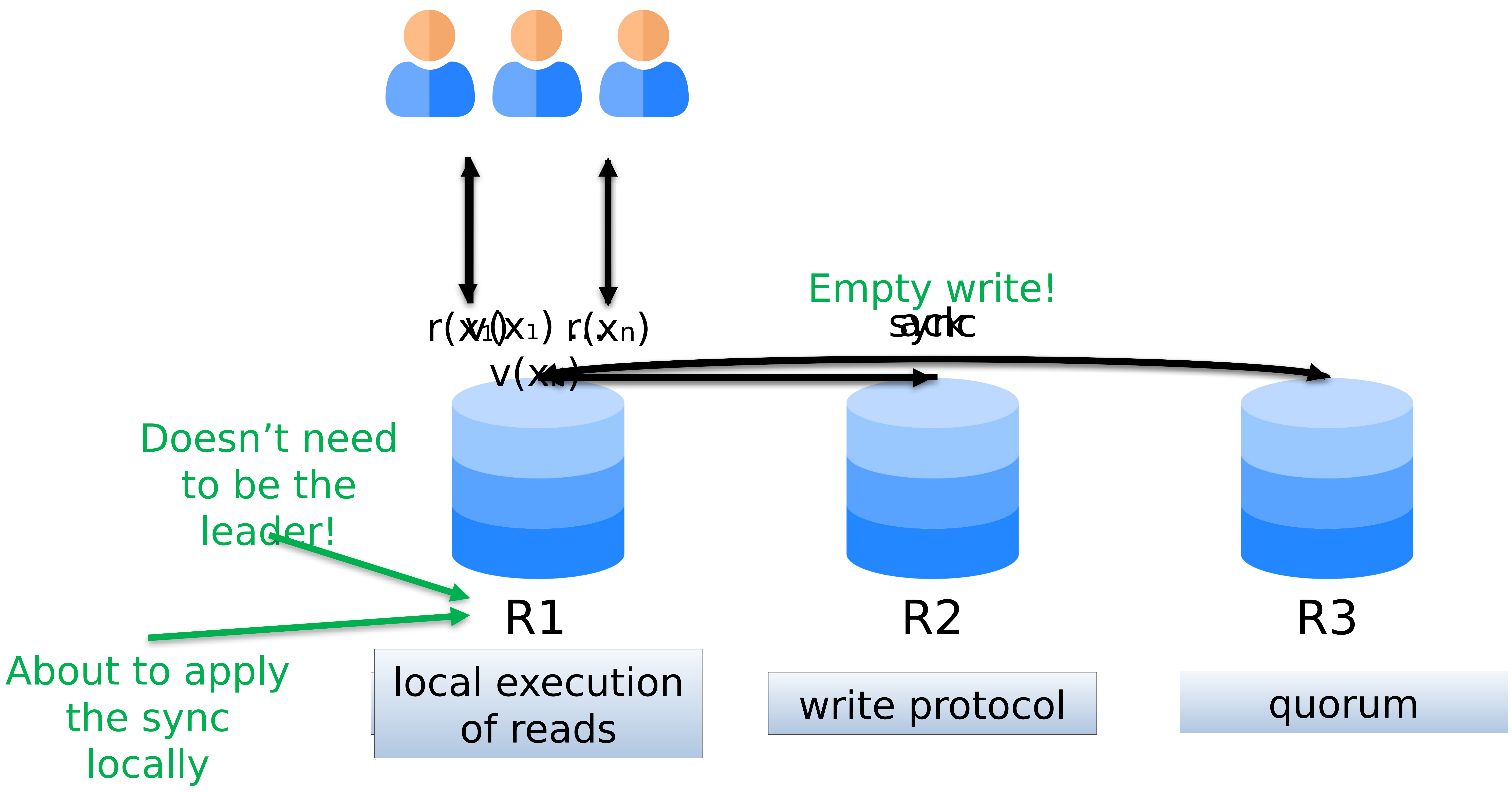
c) Once the sync is “applied locally”, it ensures that all prior writes are seen, hence subsequent local reads will be linearizable

- execute the reads in the batch

<u>Key benefit:</u>

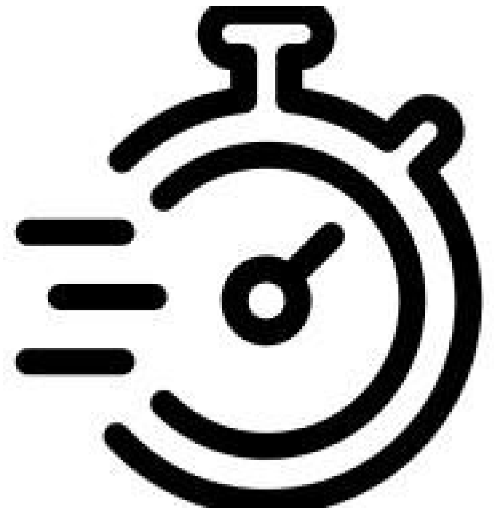
Higher throughput thanks to cheap ("almost-local”) reads

ALRs for RA protocols, visually



Further performance optimizations for ALRs

1. ALRs leverage **opportunistic batching** to not affect latency!
   - Never wait for a fixed batch size: form batches from currently pending reads.
     - Latency not affected under light load
     - Throughput maximized under heavy load



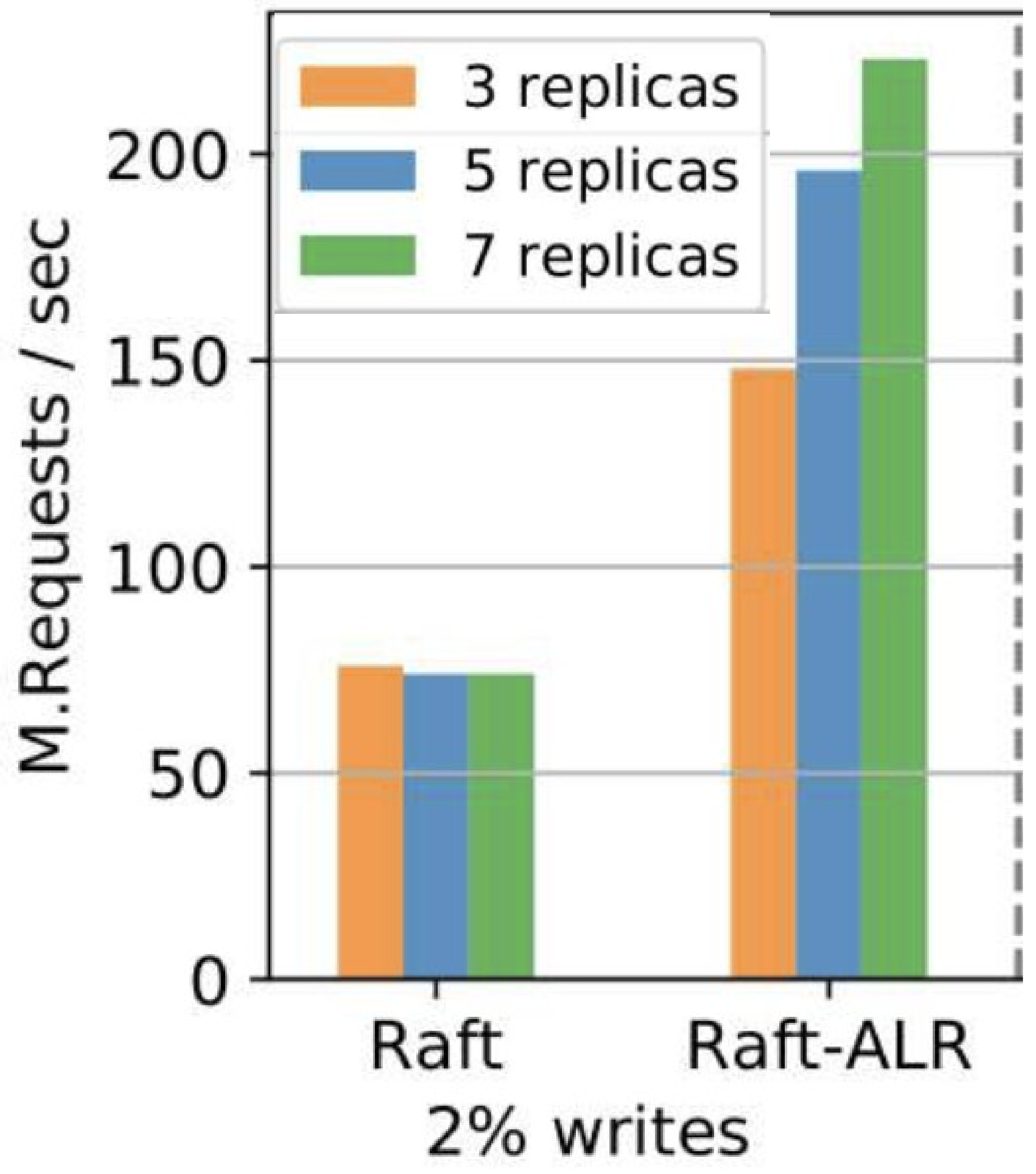
2. Writes serve as "**free**" **syncs**
   - ALRs incur **zero extra network or computation cost** on remote replicas in the presence of writes.

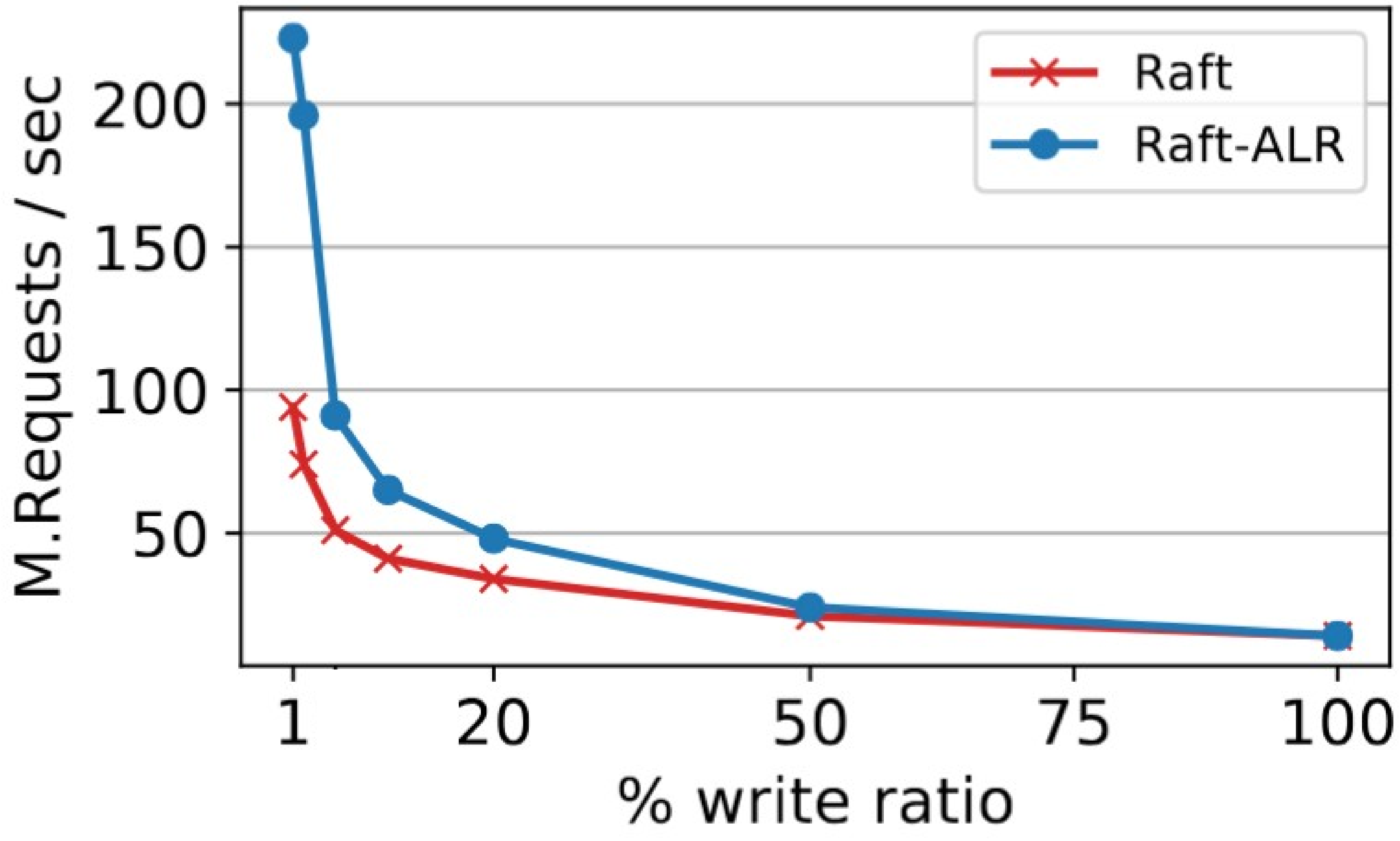
So do ALRs improve performance?

ALRs in practice: Evaluation highlights

Evaluated an ALR-enhanced variant of the state-of-art RA protocol: Raft

- Fastest RA baseline that heavily exploits traditional batching for performance.





Raft-ALR: Unlocks efficient multi-replica reads

- Significantly higher throughput: 2-3x better than baseline at 2% writes.
- More replicas ☾ higher throughput (unlike vanilla Raft)

What about other classes of protocols?

ALRs improve all classes of protocols

Linearizable

(RA) ronous

Local read

ALRs: boost throughput of RA protocols
> By enabling low-cost reads from all replicas

RA protocols: e.g. Raft, Paxos, ABD
- Remote (costly) reads
- Asynchronous
- Linearizable

RA with ALRs → Performant
RC with ALRs → Linearizable
LS with ALRs → Asynchronous

ALRs: make safe under Asynchrony the LS protocols
> By removing their dependency on time-based leases and synchronous assumptions

LS protocols: e.g. Hermes, CRAQ
- Local reads
- Synchronous
- Linearizable

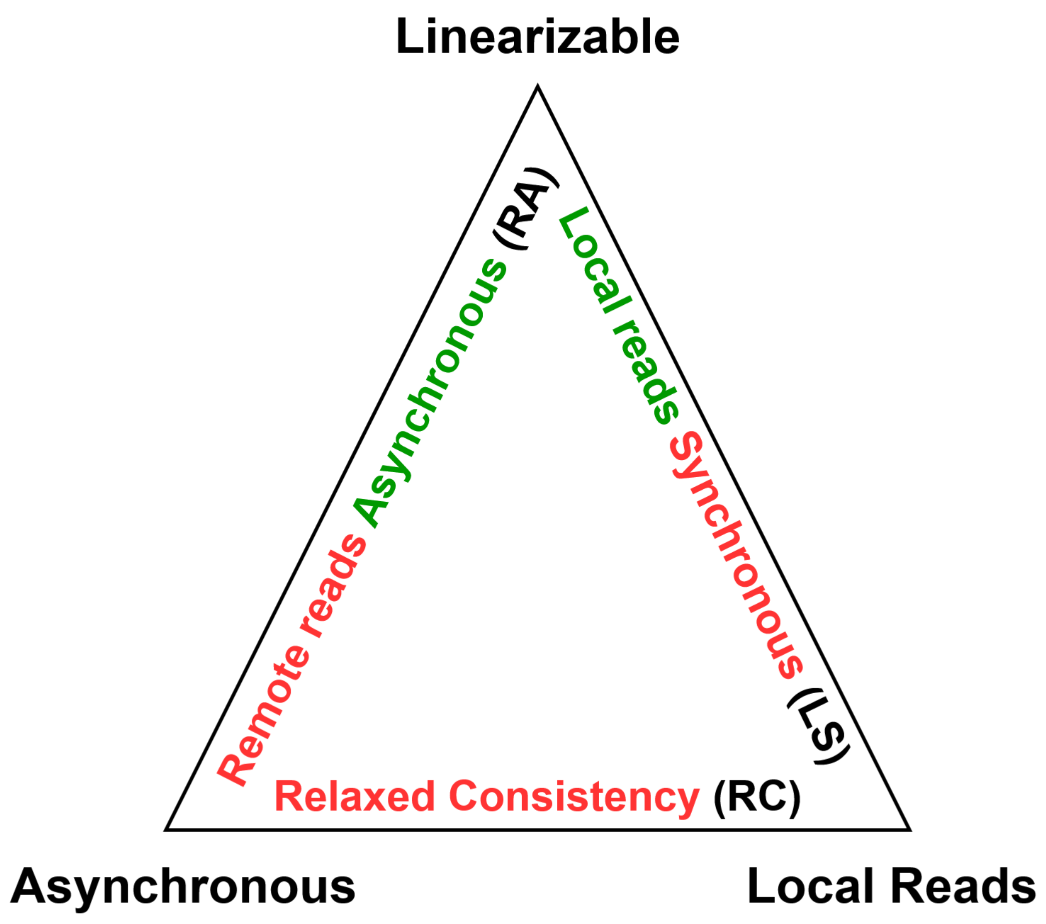
Asynchronous

Local Reads

RC protocols: e.g. ZAB
- Local reads
- Asynchronous

Fantastic! Let's summarize …

Summary



- State-of-the-art crash-tolerant protocols: 2 out of 3!
  1. Linearizability 2. Asynchrony 3. Local Reads
- We proved the **L²AW impossibility**
  Linearizable & Asynchronous crash-tolerant protocols can't have Local reads!
- Introduced Almost Local Reads (**ALRs**)
  Improve protocols in any of 3 design space corners
- ALRs exploit *opportunistic batching* & *zero-cost syncs*
  to achieve low latency and high throughput

RA with ALRs → Performant
RC with ALRs → Linearizable
LS with ALRs → Asynchronous

- *Raft-ALR (RA):* much higher throughput & better scalability
  Hermes- (LS) & ZAB-ALR (RC): linearizability under asynchrony at high throughput

More at law-theorem.com