The LAW Behind ALRs: Redefining Crash-Tolerant Reads

Antonios Katsarakis[★*], Emmanouil Giortamis[♣*], Vasilis Gavrielatos[★], Pramod Bhatotia[♣],
Aleksandar Dragojevic[♦], Boris Grot[♠], Vijay Nagarajan[♠], Panagiota Fatourou[♥]
[★]Huawei Research, [♣]TU Munich, [♦]Citadel Securities, [♠]University of Edinburgh, [♥]University of Crete and FORTH ICS, [*]Equal contribution

Today's cloud services rely on high-performance distributed datastores for storing and accessing their data. These services are often characterized by read-dominant accesses and numerous concurrent requests [1, 5]. Thus, datastores must provide high throughput to meet the performance demands of online services while offering high availability, despite being deployed on failure-prone commodity hardware [6].

Data replication is a fundamental feature of high performance and resilient datastores. Data must be replicated across multiple servers to increase throughput because a single server often cannot keep up with the request load [4]. Replication is also necessary to guarantee that a crash of a server does not render the dataset inaccessible.

Motivation. Keeping the replicas consistent, to ensure that the services running on the datastore operate correctly, is a challenge, especially in the presence of crashes. A *crash-tolerant replication protocol* is responsible for keeping the replicas of a datastore consistent – even when crashes occur – by determining the necessary actions to execute reads and writes. Several crash-tolerant protocols favor performance by relaxing consistency (***RC protocols***). As such, their reads may return stale values leading to nasty surprises for both clients and developers [15, 17]. There exist however, protocols that offer strongly consistent (i.e., linearizable) reads, which are more desirable for correctness and programmability.

Crash-tolerant protocols can be synchronous or asynchronous based on the model they rely on to ensure consistency. Synchronous protocols, which depend on known bounded processing and communication delays, are easier to design. However, in the real world, distributed datastores are deployed over complex software stacks and virtualization layers [2, 14]. Consequently, the network and compute nodes of a distributed datastore experience asynchrony and other timing anomalies, which may lead to timing violations and compromise the safety of synchronous protocols. To tolerate such timing violations, safer protocols adopt the *asynchronous* model where there are no timing assumptions, implying that processing and communication delays can be arbitrary.

Existing crash-tolerant replication protocols that afford linearizable reads fall into two categories; local reads under synchrony (***LS protocols***), and remote reads under asynchrony (***RA protocols***). LS protocols offer cheap linearizable reads that are served locally on a single replica, without inter-replica communication, as in Hermes [12]. However, these protocols assume a synchronous model [7] (e.g., to exploit leases). In contrast, RA protocols, such as Raft [16] and Paxos [13], are safe under asynchrony, but for each read, they mandate costly inter-replica coordination and the involvement of remote replicas. As Schwarzmann and Hadjistasi advocate [10], it is important to study the feasibility of crash-tolerant protocols that support linearizable local reads under asynchrony.

There exist fundamental theoretical results related to asynchronous replication, including the FLP result [8] and the CAP theorem [3, 9], but neither suffice to answer the above as both fall short in examining the performance of reads.

Theory. To address the existing gap in understanding, this work sheds light on a fundamental three-way tradeoff of crash-tolerant protocols, revealing a tension between consistency, performance, and the timing assumptions of a system. Briefly, we present **the L^{2}AW theorem**, an impossibility result which asserts that *in any **L**inearizable **A**synchronous read/write register implementation that tolerates even a single crash (**W**ithout blocking reads or writes), no reads are **L**ocal.*

We observe that the performance aspect of this tradeoff affects the latency but not necessarily the throughput of reads. Thus, asynchronous linearizable reads need not be as costly as in existing RA protocols, where each read incurs network and computation costs to remote replicas.

Practice. Capitalizing on the above insight, we introduce ***almost-local reads*** (ALRs), a scheme that affords low-cost reads in a linearizable and crash-tolerant manner under asynchrony. ALRs exploit the high volume of concurrent reads in online services by leveraging batching, but with a twist. Unlike traditional batching, all reads in an ALR-batch are executed against the local replica of a server, and only a lightweight *sync* operation per batch involves remote replicas. The sync incurs only a small network and computation cost regardless of the batch size. Moreover, the sync can sometimes be elided as existing writes can act as implicit syncs. As a result, ALRs incur little or no extra network and processing costs to remote replicas, thus achieving the performance of local reads while offering linearizability under asynchrony.

When applied to the protocols in all three corners of the design space (i.e., RA, RC, LS), ALRs add the missing piece: ❶ they improve the throughput for RA protocols (e.g., Raft) ❷ ensure linearizability for RC protocols (e.g., ZAB [11]) and ❸ they allow LS protocols (e.g., Hermes) to operate under asynchrony. Our experiments show that ALR-enhanced variants of ZAB and Hermes protocols on 95% reads come within 2% and 5% of their original throughput, respectively, while also ensuring linearizability under asynchrony. Finally, ALRs yield over 2.5× higher throughput on Raft in read-intensive workloads without sacrificing consistency or asynchrony.

[★]This work occurred when the authors were at the University of Edinburgh.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 53–64.

[2] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[3] Eric Brewer. 2000. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*. ACM, USA, 7–.

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 Conference on Annual Technical Conference (ATC'13)*. USENIX, Berkeley, 49–60.

[5] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proc. VLDB Endow.* 15, 9 (may 2022), 1965–1977.

[6] Kelly Clay. 2013. Amazon.com Goes Down, Loses $66,240 Per Minute. https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#4e849f8b495c.

[7] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382.

[9] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News* 33, 2 (2002), 51–59.

[10] Theophanis Hadjistasi and Alexander A Schwarzmann. 2018. Consistent Distributed Memory Services: Resilience and Efficiency. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[11] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance Broadcast for Primary-backup Systems. In *Proceedings of the IEEE 41st International Conference on Dependable Systems&Networks (DSN '11)*. IEEE, USA, 245–256.

[12] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217.

[13] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.

[14] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[15] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 295–310.

[16] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX, USA, 305–320.

[17] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (2009), 40–44.

Beyond reCAP: Linearizable Local Reads and Asynchronous Replication

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

† Huawei Research, ♣TU Munich, ♦Citadel Securities, ♠University of Edinburgh, ♥University of Crete & FORTH, *Equal contribution

†This work began when the author was at the University of Edinburgh and is not sponsored by Huawei.

Motivation

Online Services & Cloud Applications

Characterized by
- Many **concurrent requests**
- **Read intensive** workloads
- Need for **data reliability** → run on fault-prone h/w

Fault-tolerant Replicated Datastores

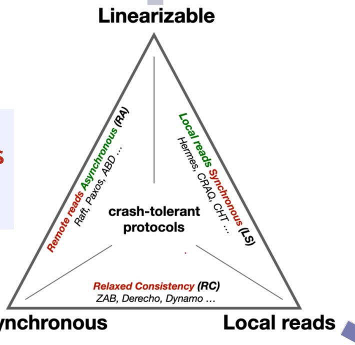
- **Crash-tolerance**: data are replicated
- **High performance**: especially for reads
- **Strong consistency** under **asynchrony** → correct — even if timeouts don't hold

Crash-tolerant Replication Protocols
determine actions for *reads* & *writes*

Ideal features
1. **Linearizable**
2. **Asynchronous**
3. **Local reads**: for max perf

Theory

Existing crash-tolerant protocols: 2 out of 3



RA protocols:
- Remote (costly) reads
+ Linearizable
+ Asynchronous

LS protocols:
- Synchronous
+ Linearizable
+ Local reads

RC protocols:
- Relaxed Consistency
+ Asynchronous + Local reads

The L²AW theorem

In any *Linearizable* *Asynchronous* read/write register implementation that tolerates even a single crash (**W**ithout blocking reads or writes), no reads are *Local*

So can we not improve read performance without compromizes?

L²AW vs. CAP

Both Linearizability & Asynchrony

L²AW read performance in its tradeoff
Key for read-dominant workloads

Fault-tolerance

CAP: network partitions
+ msg loss + partitioned nodes exec ops to violate safety

L²AW: server crashes
+ no msg loss + crashed nodes do not exec ops to violate safety

When must compromise?

CAP: during network partitions (not during partition-free) sacrifice safety or progress of ops

L²AW: always sacrifice local reads (even if crashes have not occurred)

Practice

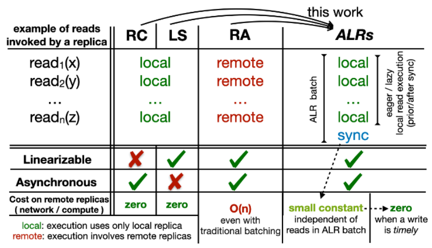
Almost Local Reads (ALRs)

Inevitably ALR latency > local reads
But **little or no extra** network and processing **costs to remote replicas**

ALRs batch reads with a twist
Exec all reads in batch w/ local replica + one sync per batch on remote nodes

Syncs are cheap!
- writes act as implicit zero-cost syncs
- explicit sync has small constant cost
- 1 sync per batch regardless its size

Add missing piece to protocols of all 3 (RC, LS, RA) categories



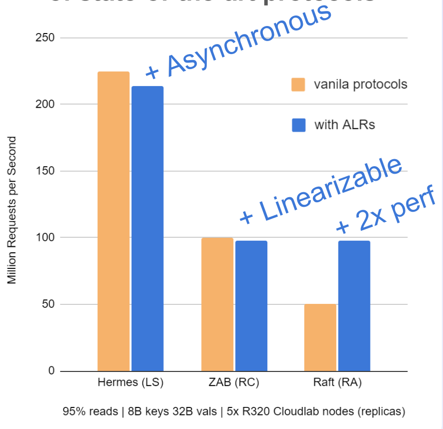
example of reads invoked by a replica	RC	LS	RA	ALRs (this work)	
read₁(x)	local	local	remote	local	
read₂(y)	local	local	remote	local	
...	...	...	...	...	
readₙ(z)	local	local	remote	local	
				sync	
Linearizable	✗	✓	✓	✓	
Asynchronous	✓	✗	✓	✓	
Cost on remote replicas (network / compute)	zero	zero	O(n) even with traditional batching	small constant independent of reads in ALR batch	zero when a write is *timely*

ALR batch
eager / lazy local read execution (prior/after sync)

local: execution uses only local replica
remote: execution involves remote replicas

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

ALR-enhanced throughput of state-of-the-art protocols



95% reads | 8B keys 32B vals | 5x R320 Cloudlab nodes (replicas)