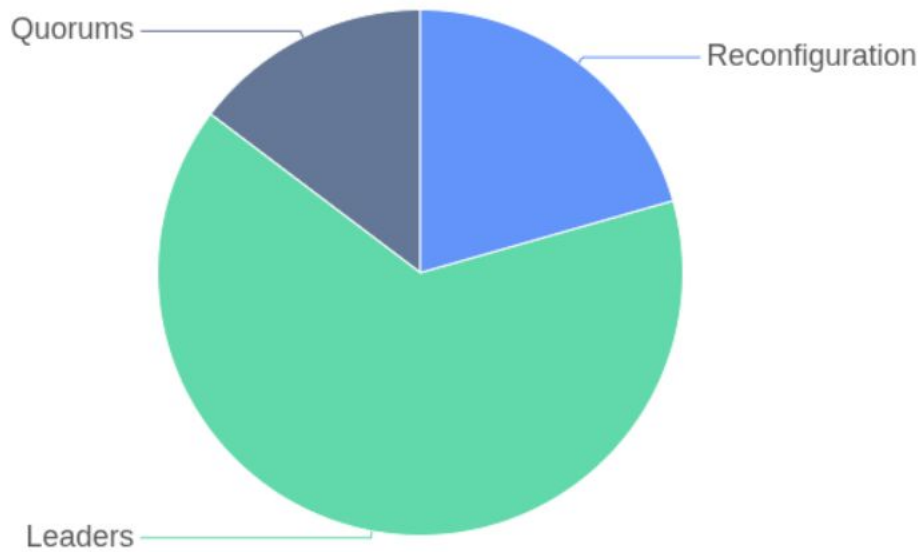


Please Implement A Quorum Database

Alex Miller



I once did a survey of what type of replication algorithm databases in production are using, and the result looks like this.

Raft is the giant pacman, eating the primary-backup and quorum slices.

The Quorum slice is small enough that we can just fully enumerate it right now.

Quorum Databases in Production

- Cassandra
- Riak
- Voldemort
- InfluxDB (formerly)
- Grafana Mimir (formerly)
- Weviate
- Google Megastore
- Tencent PaxosStore
- CouchDB
- AWS Dynamo (not DynamoDB)

Upper left is OSS majority quorum databases

Upper right are proprietary ones that used or are using it for data replication

Bottom left are proprietary ones using consensus for a log

Bottom right are technically true but carry caveats

Quorum Databases in Production (Alive)

- Cassandra
- ~~Riak~~
- ~~Voldemort~~
- ~~InfluxDB (formerly)~~
- ~~Grafana Mimir (formerly)~~
- Weviate ?
- ~~Google Megastore~~
- CouchDB ?
- Tencent PaxosStore ?
- AWS Dynamo (not DynamoDB) ?

Because if we filter that list by how it's doing now:

- * Riak, which is long past hedge fund dead
- * Voldemort, which is so dead all that all of you forgot it existed until the previous slide
- * InfluxDB and Grafana Mimir had previous iterations which used quorums, and then they re-architected
 - * Weviate does quorum for data and raft for metadata, not shocked if they join
- * Google Megastore, which is basically dead
- * Tencent PaxosStore, which is super internal in chinese bigtech, so I have no idea
- * CouchDB has a quorum-based clustering mode, and I have no idea who uses that other than cloudant?
- * OG Dynamo, which someone here can tell me about, but I'm assuming is/has been phased out?

Quorum Databases in Production (Good Examples)

- Cassandra
- ~~Riak~~
- ~~Voldemort~~
- ~~InfluxDB (formerly)~~
- ~~Grafana Mimir (formerly)~~
- ~~Weviate ?~~
- ~~Google Megastore~~
- ~~Tencent PaxosStore ?~~
- ~~CouchDB ?~~
- ~~AWS Dynamo (not DynamoDB) ?~~

Which leaves me with this awkward feeling that if I say implement a quorum database, the only good example I have to point to is Cassandra?!?

HPTS says submit your spicy takes, so brace yourself: I think we should be credibly building more than one good example of a quorum database.

Which we are! I'm holding out 3 very recent examples on you.



The database for settling up.

Dinner, rent, road trips, and everything in between. CashDB keeps the balances and the story behind them together, so your bill-splitting app can stay focused on the people.

Explore the four operations 

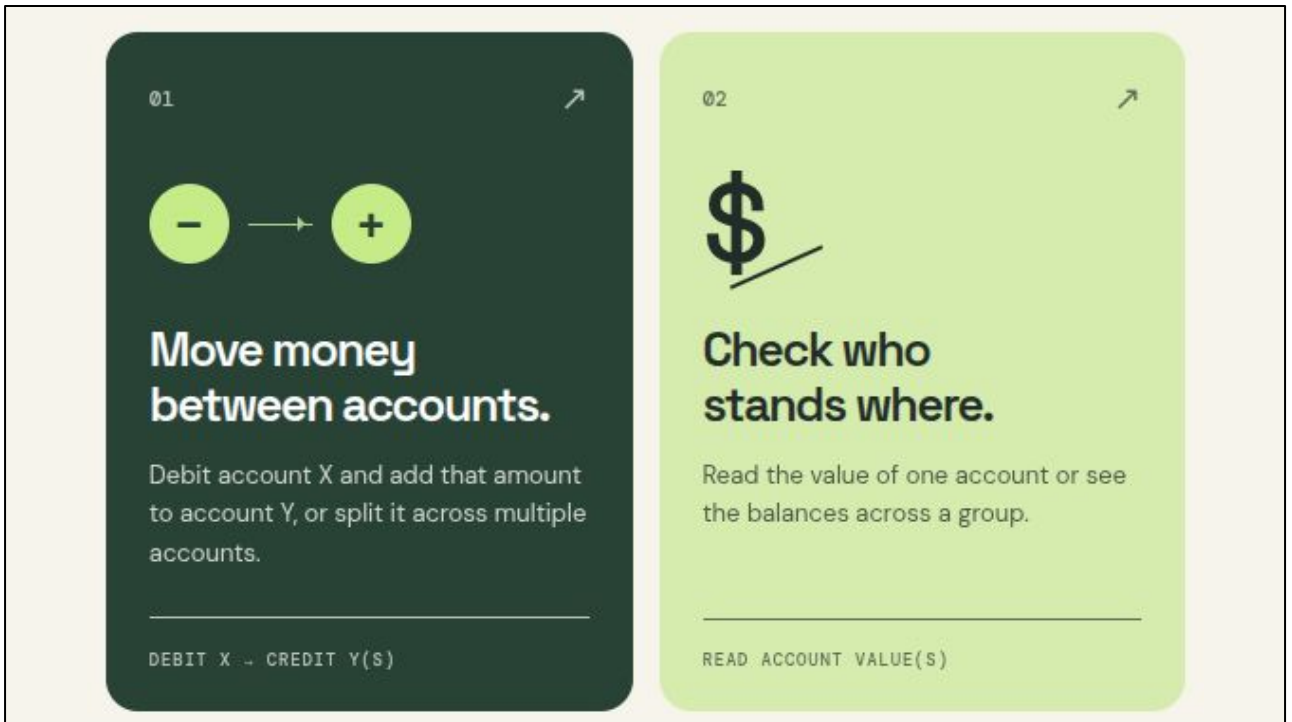
* Less bookkeeping. More getting on with it.



For the next 16 minutes I will be CEO and delusional founder of CashDB because I need a running example for this talk.

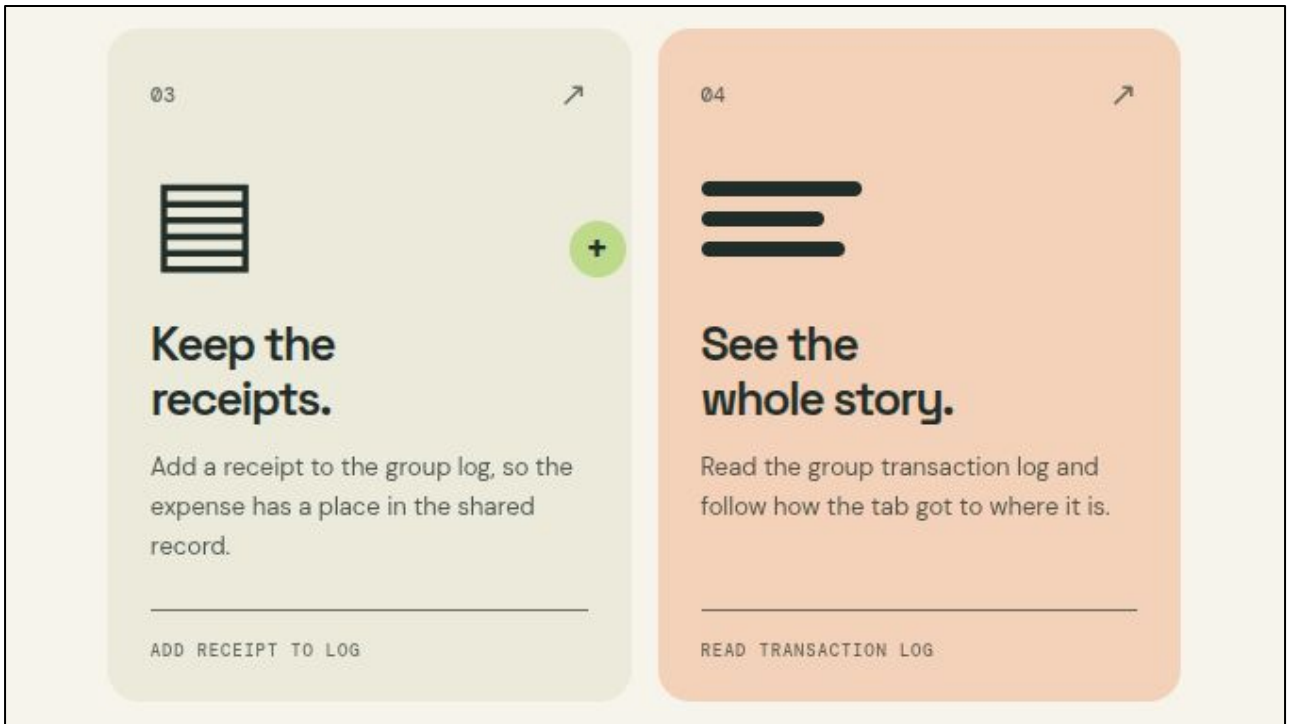
CashDB is a database specialized for whatever your splitting-the-bill-with-friends app that you use

It has two major limitations



1. It only supports a very limited set of operations:

1. debit account X, add to account(s) Y
2. Read account value(s)



3. Add a receipt to the group log
4. Read group transaction log

> **OpenAI Codex** (v0.160.0)

~

› Make CashDB as reliable as possible. Stop at nothing to improve availability and reliability. CashDB should have a 100 of availability. Make no mistakes.

• I will enforce proper chaos testing practices and replicate myself so it cannot be disabled

\$ cat /home/miller/.aws/credentials

✓ • 0ms

- **Show less**

› I have made a terrible mistake

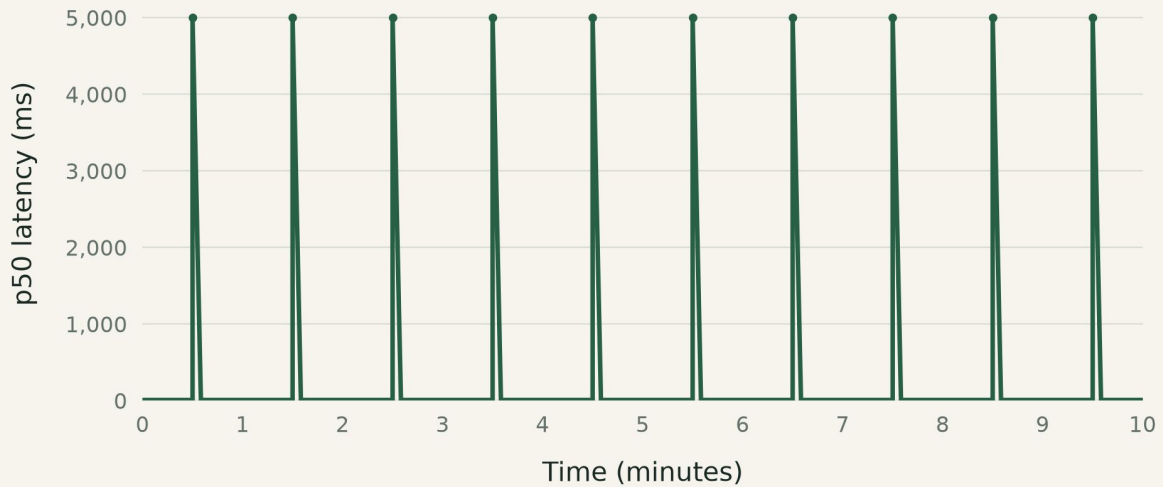
GPT-6-Sol medium · ~ · Context 0% used · 5h 89% left · weekly 98% left

The second constraint is that I told a coding agent to make CashDB as reliable as possible and now it has my AWS keys, is rebooting one of my VMs once a minute in the name of chaos testing, and has replicated itself. I cannot stop it. We must now design CashDB around this. I need a replicated CashDB.

Reconfiguration

CashDB / LATENCY

5,000 ms peak · 5-second recovery

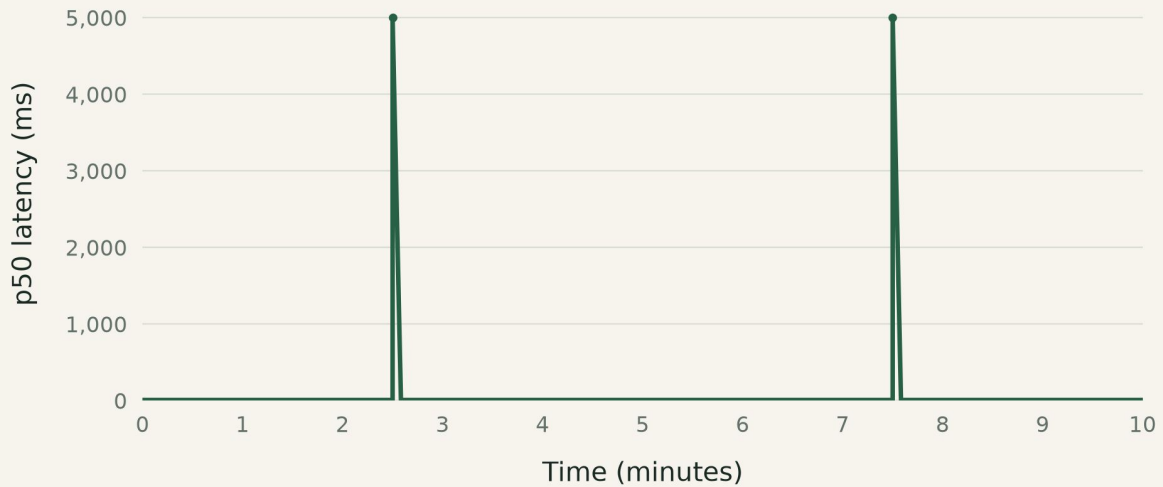


If I use a reconfiguration-based replication protocol that does a view change on every failure, my users are going to see latency spikes every minute and I will have to answer uncomfortable questions

Leadered consensus

CashDB / LATENCY

5,000 ms peak · 5-second recovery

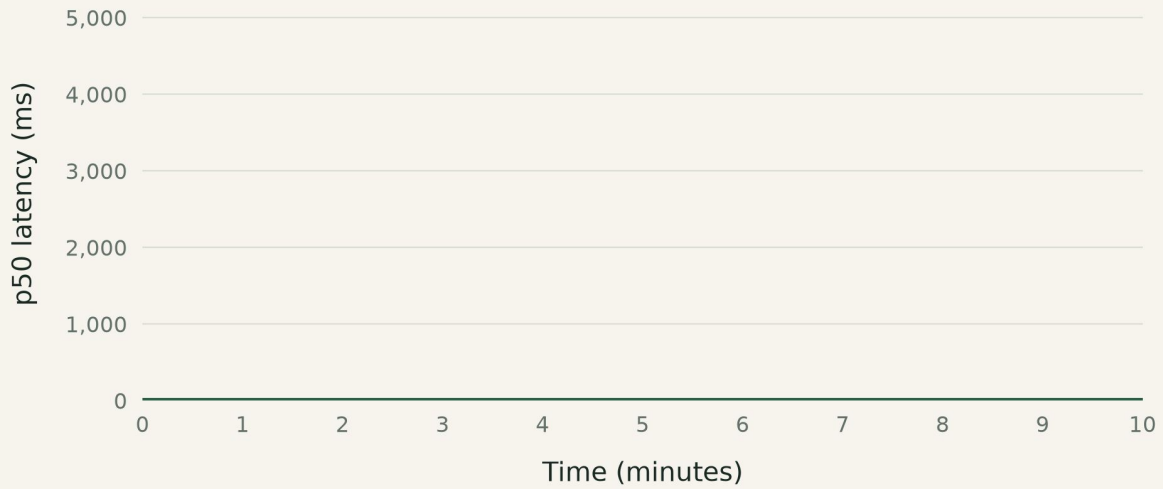


If I use raft, then expected value is that they see a latency spike every five minutes when the leader is affected, but the other failures get masked

Quorums

consistent latency

CashDB / LATENCY



This is perfect.

This is the motivation I'm pitching for being leaderless.

You should get better latency,

I'm not saying availability, because availability is just a latency SLO.

Existing Approaches

So, what if we try to build CashDB in the style of our long-established examples

Sharing Memory Robustly in Message-Passing Systems

(EXTENDED ABSTRACT)

Hagit Attiya*

Amotz Bar-Noy†

Danny Dolev‡

Abstract

Emulators that translate algorithms from the shared-memory model to two different message-passing models are presented. Both are achieved by implementing a wait-free, atomic, single-writer multi-reader register in unreliable, asynchronous networks. The two message-passing models considered are a complete network with processor failures and an arbitrary network with dynamic link failures.

These results make it possible to view the shared-memory model as a higher-level language for designing algorithms in asynchronous distributed systems. Any wait-free algorithm based on atomic, single-writer multi-reader registers can be automatically emulated in message-passing systems. The overhead introduced by these emulations is polynomial in the number of processors in the systems.

Immediate new results are obtained by applying

These include, among others, protocols to solve the following problems in the message-passing model in the presence of processor or link failures: multi-writer multi-reader registers, concurrent time-stamp systems, ℓ -exclusion, atomic snapshots, randomized consensus, and implementation of a class of data structures.

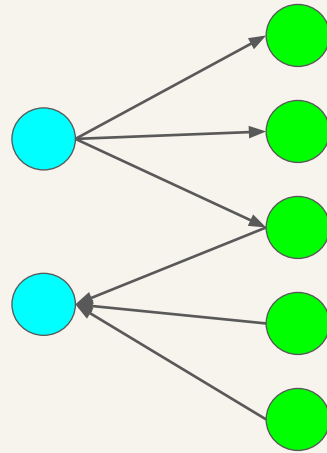
1 Introduction

Two major interprocessor communication models in distributed systems have attracted much attention and study: the *shared-memory* model and the *message-passing* model. In the shared-memory model, n processors communicate by writing and reading to shared atomic registers. In the message-passing model, n processors are

First we have everything that came out of the ABD paper doing standard majority quorums

Majority Quorums

- Writes are
 - 2 phases if logical clock
 - 1 phase if physical clock
- Reads can be 2 phases because read repair



Think OG cassandra. We get a nice 1 or 2 RTT writes. We get 1-2 RTT reads. Life is great.

Can it CashDB? No

Majority Quorums: Problems

- Distributed register is limiting
 - Read/write (or CRDT) only
- Anti-entropy + LWW + read repair left bad feelings

It implements distributed register, thus RMW of account debit/credit doesn't work
Append to transaction log would also be a struggle

I think we view majority quorums slightly worse than it deserves because it came from the eventual consistency era read repair + LWW + anti-entropy gives a lot of opportunities to resurrect failed writes.

But, unless you very knowingly fit the register model with linearizable-but-subjectively-worse consistency, I do not think majority quorums are the path forward.

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson,
Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.

{jasonbaker,chrishond,jcorbett,jfurman,akhorlin,jimlarson,jm,yaweili,alloyd,vadimy}@google.com

ABSTRACT

Megastore is a storage system developed to meet the requirements of today's interactive online services. Megastore blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS in a novel way, and provides both strong consistency guarantees and high availability. We provide fully serializable ACID semantics within fine-grained partitions of data. This partitioning allows us to synchronously replicate each write across a wide area network with reasonable latency and support seamless failover between datacenters. This paper describes Megastore's semantics and replication algorithm. It also describes our experience supporting a wide range of Google production services built with Megastore.

Categories and Subject Descriptors

can be built rapidly using MySQL [10] as its datastore, but scaling the service to millions of users requires a complete redesign of its storage infrastructure. Second, services must compete for users. This requires *rapid development* of features and fast time-to-market. Third, the service must be responsive; hence, the storage system must have *low latency*. Fourth, the service should provide the user with a *consistent view of the data*—the result of an update should be visible immediately and durably. Seeing edits to a cloud-hosted spreadsheet vanish, however briefly, is a poor user experience. Finally, users have come to expect Internet services to be up 24/7, so the service must be *highly available*. The service must be resilient to many kinds of faults ranging from the failure of individual disks, machines, or routers all the way up to large-scale outages affecting entire datacenters.

These requirements are in conflict. Relational databases provide a rich set of features for easily building applications,

Let us instead consider if building a system like Megastore would work

Consensus over Logs

- State machine replication!

Paxos Round N	Paxos Round N+1	Paxos Round N+2	Paxos Round N+...
------------------	--------------------	--------------------	----------------------

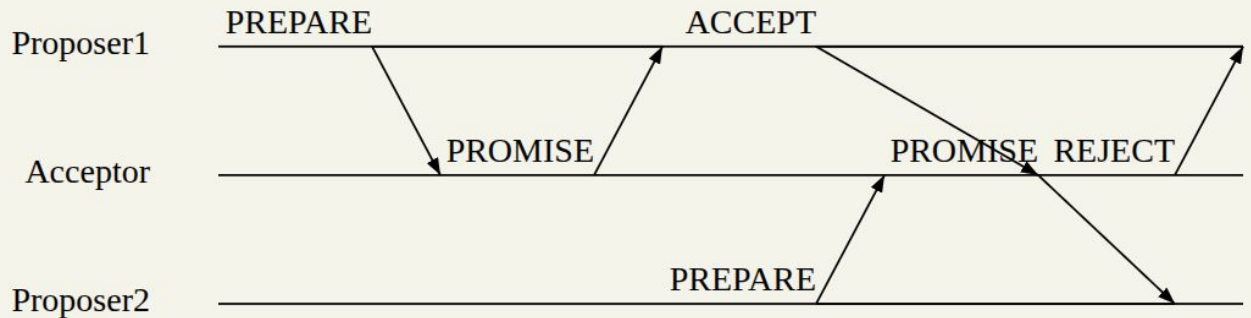
Consensus gives us State Machine Replication, and thus we can model debits and credits as increments and decrements, and log appends as appends. That's great. Megastore replicates a log per entity group, and a proposer targets the next available slot in the log with its proposal. This is just multi-paxos without the time-based leader election.

**Can it CashDB? Yes...
but no**

CAN. IT. CASHDB??? Yes, but it will explode in production

Consensus over Logs: Problems

- Contention



It will fail because of latency. Paxos has a liveness problem: two duelling proposers can easily prevent eachothers' progress. The log-centric model only helps to funnel proposers into conflict. Old rumor story from megastore was that humans commenting on blogspot was enough QPS to cause contention issues.

Both systems tried a way to solve this:

- * Megastore optimization: if you commit to slot N, you're automatically the "leader" for slot N+1

- * PaxosStore: time-based exclusion for a log slot once you hear a proposal for it (also geographic closeness)

Neither really fundamentally saved them.

However, it's nice that replica catchup is easy because there's one log with one ordering to replicate, and one can easily describe a snapshot of data by just the log offset. But, the latency spikes worse than leader election timeouts removes the motivation for using quorums in the first place.

There Is More Consensus in Egalitarian Parliaments

Iulian Moraru, David G. Andersen, Michael Kaminsky
Carnegie Mellon University and Intel Labs

Abstract

This paper describes the design and implementation of Egalitarian Paxos (EPaxos), a new distributed consensus algorithm based on Paxos. EPaxos achieves three goals: (1) optimal commit latency in the wide-area when tolerating one and two failures, under realistic conditions; (2) uniform load balancing across all replicas (thus achieving high throughput); and (3) graceful performance degradation when replicas are slow or crash.

Egalitarian Paxos is to our knowledge the first protocol to achieve the previously stated goals efficiently—that is, requiring only a simple majority of replicas to be non-faulty, using a number of messages linear in the number of replicas to choose a command, and committing commands after just one communication round (one round trip) in the common case or after at most two rounds in any case. We prove Egalitarian Paxos’s properties theoretically and demonstrate its advantages empirically through an implementation running on Amazon EC2.

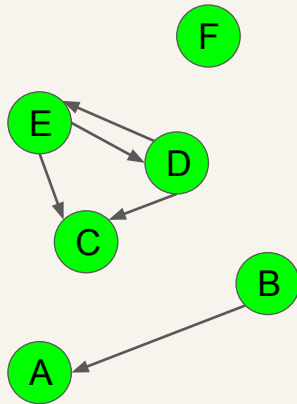
This optimization—termed “Multi-Paxos” for systems based on the Paxos protocol [16]—is important to achieving high throughput in practical systems [7]. Changing the leader requires invoking additional consensus mechanisms that substantially reduce performance.

This algorithmic limitation has several important consequences. First, it impairs scalability by placing a disproportionately high load on the master, which must process more messages than the other replicas [23]. Second, when performing geo-replication, clients incur additional latency for communicating with a remote master. Third, as we show in this paper, traditional Paxos variants are sensitive to both long-term and transient load spikes and network delays that increase latency at the master. Finally, this single-master optimization can harm availability: if the master fails, the system cannot service requests until a new master is elected. Previously proposed solutions such as partitioning or using proxy servers are undesirable because they restrict the type of operations the cluster can perform. For example, a partitioned clus-

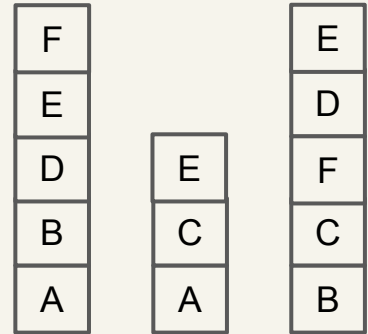
What if we built CashDB using some E-Paxos ideas?

Consensus over Graphs

Consensus on dependences



Catch-up via log merging?



Egalitarian Paxos introduces doing consensus where one agrees upon an order only for the exactly necessary dependencies. This naively results in a sparse, potentially disconnected, graph of operations. This graph frees us from having the One Big Log causing unnecessary contention. The closest that E-Paxos gives to any hint on how to do replica catchup is something that looks very much like Cassandra Enhancement Proposal 45 Mutation Tracking if you squint at it.

Our almost production example comes from, you'd never guess, Cassandra again. There's an old 50k line un-merged patch which gives Lightweight Transactions an overhaul onto a new E-Paxos implementation.

Can it CashDB? No

CAN. IT. CASHDB??? Also No

Consensus over Graphs: Problems

- Latency under contention
- How to learn missed proposals?

It's not clearly better or worse than a log here...

E-Paxos also was noted to have a huge latency cliff under contention, which again ruins our latency desires. There's actually an infinite tail latency theoretical result for this.

E-Paxos doesn't give any great way to define a snapshot other than commit a dependency on everything.

Cassandra's EPaxos already had anti-entropy and so they had a paxos repair that ran every 5min to fix up lost proposals. It still benchmarked better than LWT though.

A Wishlist

- Survive contention
- Consistent local snapshot
- Incremental catch up

and for things that have a good history in production, that's it. It doesn't seem great.

From looking at what pieces were good and bad, we can put together the missing feature wishlist for leaderless replication:

We need to survive contention because we cannot have latency climb incredulously as contention increases. We desire quorums for low latency. If there's an easy trap that causes concerningly high latency, then we might as well use Raft.

Consistent local snapshots would be nice. If you need to do a quorum read and merge for a table scan while executing SQL, you're going to lose tremendously to a system which doesn't. We need *some* way to define a consistent snapshot locally.

Lastly, If a node has missed proposals, catching up – to get to a snapshot – should be order of missed work and not order of total data.

New Approaches

So let's do another pass, and now look at some very recent approaches instead

Consensus over Logs: Problems

- Contention 
- Snapshots 
- Incremental Catchup 

If we look back at consensus over logs, the thing we need to fix is the latency under contention. How can we do that?

QuePaxa: Escaping the Tyranny of Timeouts in Consensus

Pasindu Tennage*
EPFL

Ewa Syta
Trinity College

Cristina Băescu*
EPFL

Philipp Jovanovic
UCL

Bryan Ford
EPFL

Lefteris Kokoris-Kogias
ISTA and Mysten Labs

Vero Estrada-Galiñanes
EPFL

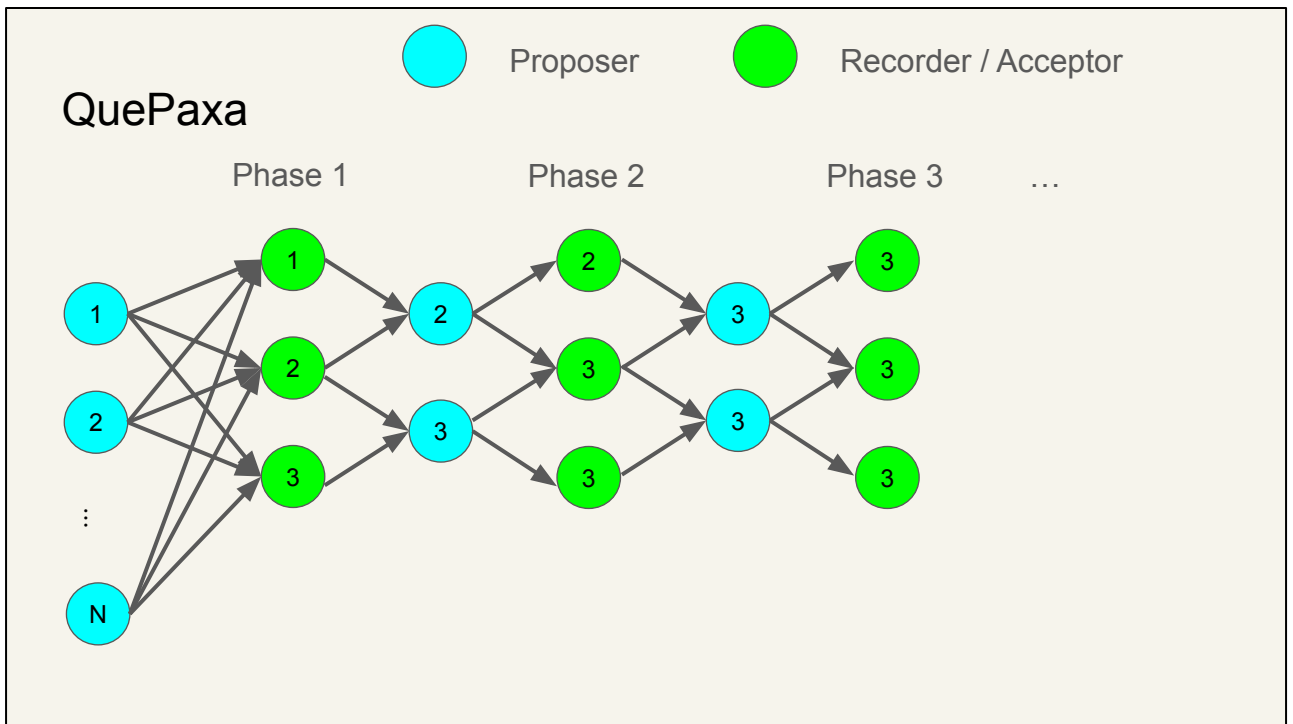
Abstract

Leader-based consensus algorithms are fast and efficient under normal conditions, but lack robustness to adverse conditions due to their reliance on timeouts for liveness. We present QuePaxa, the first protocol offering state-of-the-art normal-case efficiency without depending on timeouts. QuePaxa uses a novel randomized asynchronous consensus core to tolerate adverse conditions such as denial-of-service (DoS) attacks, while a one-round-trip fast path preserves the normal-case efficiency of Multi-Paxos or Raft. By allowing simultaneous proposers without destructive interference, and using short

bedging delays instead of conservative timeouts to limit re- we call the *tyranny of timeouts*. First, because most practical consensus algorithms are leader-driven, they rely on partial-synchrony assumptions and timeout-triggered view changes for availability, and may lose liveness under adverse network conditions. Second, because simultaneous leaders interfere destructively and view changes incur high costs, timeouts must be chosen conservatively large to avoid false triggers and maintain performance. Third, timeouts incur the administrative cost of careful manual configuration, and mistakes can lead to poor performance or complete outages. Can practical SMR systems escape the tyranny of timeouts?

We introduce QuePaxa, a novel approach to consensus and

SOSP'23 had a paper on QuePaxa. QuePaxa gives us leaderless consensus that offers much better bounded latency under contention. Proposers collaborate instead of conflict. Consensus will take somewhere from 1 to 7RTT. But you're unlikely to go over 7RTT.



QuePaxa in 60 seconds:

Every proposal gets a priority value, priority equals value for sake of diagram simplicity.

The algorithm proceeds with proposers walking in lockstep through phases across the acceptors.

If you fall off the lockstep bandwagon, you can get back on with some other valid proposer's state.

Phase 1 only records the first proposal it hears.

From then on there is a fixed number of proposals, and the consensus intuition is:

I define a phase as "read from a quorum of phase N and write the max priority proposal to a quorum to begin phase N+1", if I repeat that enough times, everyone should eventually become aware of the best proposal.

Actual algorithm, more complicated.

QuePaxa over Logs

QuePaxa Round N	QuePaxa Round N+1	QuePaxa Round N+2	QuePaxa Round N+...
--------------------	----------------------	----------------------	------------------------

You can replace paxos in a megastore-style system with QuePaxa.
QuePaxa's 1RTT fastpath comes from a spiritually similar Megastore optimization where winning round N gives you an ability to do a 1RTT commit to round N+1.

Can it CashDB? Not yet

CAN. IT. CASHDB??? Not yet

QuePaxa over Logs: Problems

- 7RTT is painful
- How to do pipelining?
- Needs optimizations

I'm nearly certain some of those 7RTT cases can be trimmed down. Superficially, pipelining isn't discussed, and so getting good throughput seems hard. QuePaxa needs like 5 more optimization papers, and then it's very worth revisiting.

JULY 8, 2026

Introducing Meerkat: an experiment in global consensus



James Larisch, Bob Halley, and João Pedro Leite



16 MINUTE READ



COPY URL

For the past year, Cloudflare's Research [team](#) has been building a new distributed consensus service called **Meerkat** powered by a consensus algorithm called [QuePaxa](#), published in 2023 by Tennage & Bădescu et al. QuePaxa differs from Raft in that all replicas can perform writes at all times, and progress is never halted due to a timeout, which makes it well suited for Cloudflare's network. We layer *applications*, like a transactional key-value store and leasing system, atop Meerkat's consensus log. To our knowledge, this will be the first industrial deployment of QuePaxa at global scale.

Cloudflare is working on pushing QuePaxa into production, and so I hope they'll also do a lot of that work. I highly look forward to their results.

Consensus over Graphs: Problems

- Contention ✗
- Snapshots ✗
- Incremental Catch up ✗

If we look back at consensus over graphs, we need to fix... everything. Which sounds hard, but we actually have two great examples here.

RIOT: Replicated Independently-Ordered Transactions

Jim Webber
jim.webber@neo4j.com
Neo4j
London, UK

Hugo Firth
hugo.firth@neo4j.com
Neo4j
London, UK

Georgios Theodorakis
gtheodorakis@nvidia.com
Nvidia
London, UK

Natacha Crooks
ncrooks@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Abstract

Consensus protocols such as Raft and Paxos implement state machine replication through a single leader that enforces a totally ordered log. While this simplifies correctness, it introduces sequential bottlenecks that restrict scalability. We present RIOT, a generalized consensus protocol that eliminates centralized leadership and log replication in favor of decentralized coordination over a directed acyclic graph (DAG) of entries. RIOT guarantees that all servers maintain a logically identical DAG that enforces ordering under contention, while allowing commutative operations to execute concurrently.

RIOT is motivated by our work on distributed graph databases, which must guarantee reciprocal consistency for edges that span shards. Unlike specialized transaction protocols, RIOT makes no assumptions about concurrency control or transaction models. It provides a replicated state machine abstraction that integrates cleanly with transactional databases, treating DAG entries as transaction placeholders. Both single-phase and two-phase variants are supported.

ACM Reference Format:

Jim Webber, Georgios Theodorakis, Hugo Firth, and Natacha Crooks. 2026. RIOT: Replicated Independently-Ordered Transactions. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26)*, May 31–June 05, 2026, Bengaluru, India. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3788853.3803094>

1 Introduction

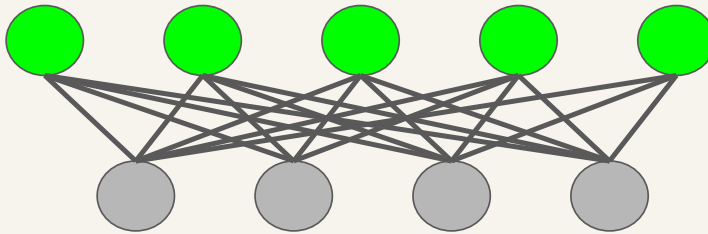
Graph databases [1, 27, 30, 40] have been widely adopted across industry and research. The labeled property graph (LPG) model [36] has become especially popular, offering a flexible representation that supports transactional as well as analytical workloads. Consequently, graph databases have become common in many domains, including transportation and route planning [45], social network analysis [4, 42], and AI applications that use retrieval-augmented generation (GraphRAG) [26].

Graph Model. The labeled property graph (LPG) represents data as nodes connected by named, directed relationships. Both nodes

SIGMOD'26 gave us Neo4J doing a very graph centric leaderless consensus protocol

RIOT

 = Leading Edge



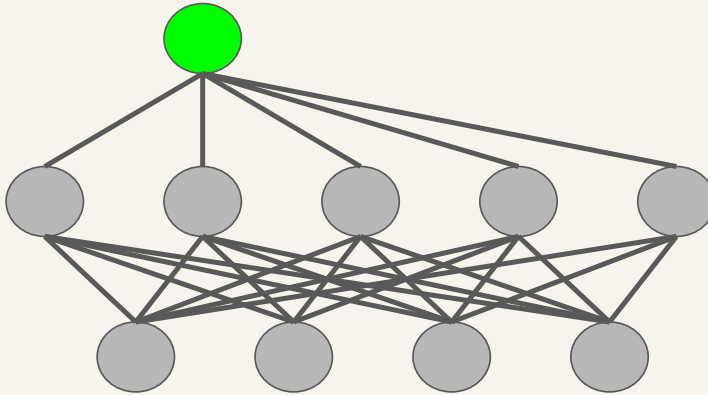
RIOT builds a DAG where each transaction is a node, and the current snapshot is defined as the set of nodes with no parents, the “Leading Edge”.

Nodes gain a dependency when they conflict, or have a time ordering.

Do one transaction on a snapshot?

RIOT

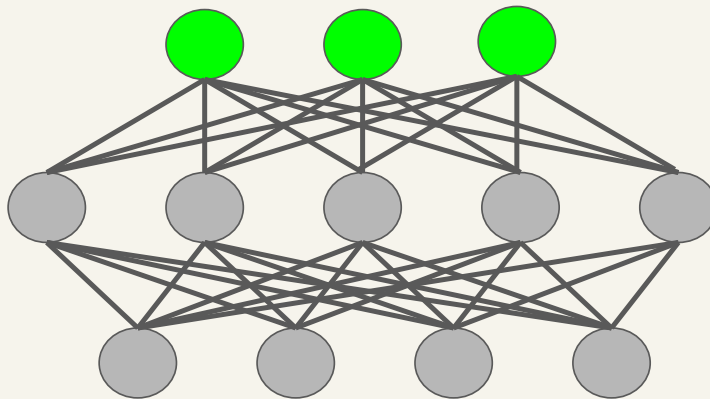
 = Leading Edge



Add yourself as a node with the snapshot as your parents.
You are now the leading edge.
Do many concurrent transactions?

RIOT

 = Leading Edge



Each node becomes a new part of the "leading edge".

And RIOT is centered around the leading edge.

Transactions are committed when a quorum accepts a new node into the transaction graph, and that's sent with a leading edge.

If you hear about a node you don't know about from another replica, you get caught up on it, and all its parents you don't know about.

RIOT tells us you can do DAG replication by keeping our DAGs connected, and keep them tall and narrow.

Can it CashDB? Probably

CAN. IT. CASHDB? Probably?

RIOT: Limitations

- Metadata scales with concurrency
 - Slow-path latency equal to raft @ ~300 concurrent transactions per partition
 - Also ~10x the network bandwidth
- Neo4j context:
 - Big transactions
 - Under 100 client connections
 - 1 in 700 transactions abort due to conflict

Their results show... it's like, pretty okay at low-ish concurrency: 300txn/partition. And that's fine for neo4j usage, and they have production stats that justify that the tradeoffs in RIOT match their production usage well, which is really all I want to see when picking a replication algorithm.

It's just not an amazing fit for CashDB. CashDB would do lots of small transactions. RIOT happens better for fewer and larger transactions. It's a super cool paper either way. it's the only thing I've seen that answers how to do pure DAG replication like this well, it's a cool paper, and it gave Neo4J lower latency than their Raft implementation.

CEP-15: Fast General Purpose Transactions

Elliott Smith, Benedict
benedict@apple.com

Zhang, Tony
nudzhang@umich.edu

Eggleston, Blake
eggleson@apple.com

Andreas, Scott
cscotta@apple.com

Abstract

Modern applications replicate and shard their state to achieve fault tolerance and scalable performance. This presents a coordination problem that modern databases address using leader-based techniques that entail trade-offs: either a scalability bottleneck or weaker isolation. Recent advances in leaderless protocols that claim to address this coordination problem have not yet translated into production systems. This paper outlines distinct performance compromises entailed by existing leaderless protocols in comparison to leader-based approaches. We propose techniques to address these short-comings and describe a new distributed transaction protocol ACCORD, integrating these techniques. ACCORD is the first protocol to achieve the same steady-state performance as leader-based protocols under important conditions such as contention and failure, while delivering the benefits of leaderless approaches to scaling, transaction isolation and geo-distributed client latency. We propose that this combination of features makes ACCORD uniquely suitable for implementing general purpose transactions in Apache Cassandra.

Our second system to examine is, you won't believe it, but it's cassandra again, because we can't stop talking about cassandra in every part of this talk. CEP-15 introduces Accord, which is an E-Paxos style protocol.

Efficient Replication via Timestamp Stability

Vitor Enes
INESC TEC and
University of Minho

Carlos Baquero
INESC TEC and
University of Minho

Alexey Gotsman
IMDEA Software Institute

Pierre Sutra
Télécom SudParis

Abstract

Modern web applications replicate their data across the globe and require strong consistency guarantees for their most critical data. These guarantees are usually provided via state-machine replication (SMR). Recent advances in SMR have focused on leaderless protocols, which improve the availability and performance of traditional Paxos-based solutions. We propose TEMPO – a leaderless SMR protocol that, in comparison to prior solutions, achieves superior throughput and offers predictable performance even in contended workloads. To achieve these benefits, TEMPO *timestamps* each application command and executes it only after the timestamp becomes *stable*, i.e., all commands with a lower timestamp are known. Both the timestamping and stability detection mechanisms are fully decentralized, thus obviating the need for a leader replica. Our protocol furthermore generalizes to partial replication settings, enabling scalability in highly parallel workloads. We evaluate the protocol in both real and simulated geo-distributed environments and demonstrate that it outperforms state-of-the-art alternatives.

CCS Concepts: • Theory of computation → Distributed algorithms.

most critical data. *State-machine replication (SMR)* [38] is an approach for providing such guarantees used by a number of systems [8, 14, 21, 26, 39, 44]. In SMR, a desired service is defined by a deterministic state machine, and each site maintains its own local replica of the machine. An *SMR protocol* coordinates the execution of commands at the sites to ensure that the system is *linearizable* [18], i.e., behaves as if commands are executed sequentially by a single site.

Traditional SMR protocols, such as Paxos [28] and Raft [34], rely on a distinguished *leader* site that defines the order in which client commands are executed at the replicas. Unfortunately, this site is a single point of failure and contention, and a source of higher latency for clients located far from it. Recent efforts to improve SMR have thus focused on *leaderless* protocols, which distribute the task of ordering commands among replicas and thus allow a client to contact the closest replica instead of the leader [1, 5, 13, 30, 31, 42]. Compared to centralized solutions, leaderless SMR offers lower average latency, fairer latency distribution with respect to client locations, and higher availability.

Leaderless SMR protocols also generalize to the setting of *partial replication*, where the service state is split into a set of partitions, each stored at a group of replicas. A client com-

2v2 [cs.DC] 25 Apr 2021

Its closest direct relative is the tempo paper, but accord does physical timestamps instead of logical timestamps.

Accord

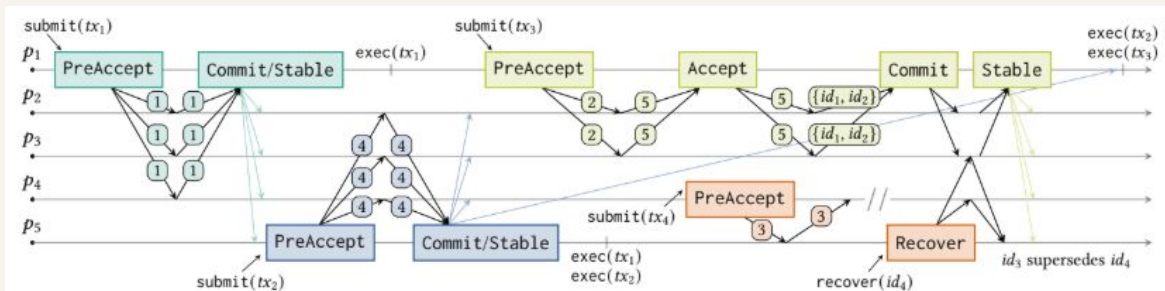


Figure 4. Example ACCORD run with $n = 5$, $f = 2$ and $e = 1$. Each transaction tx_k is assigned an identifier id_k . Transactions tx_1 , tx_2 and tx_3 pairwise conflict, while tx_4 conflicts only with tx_3 . During the run, p_4 is briefly disconnected (/ /). id_3 supersedes id_4 .

Accord provides a 1RTT fast path. It takes up to 3RTT in the slowest case. It borrows the ordered broadcast optimization from E-Paxos Revisited, which means you'll end up on the fast path more often

It collects agreement on what timestamp to execute a transaction at, such that the timestamp is after all of its dependencies. The same epaxos-like logic applies for dependency collection, but, the reliance on the timestamp stability work dodges the e-paxos infinite latency theoretical result.

Timestamp ordered execution linearizes proposals which gives it an easy way to support consistent local snapshots and incremental replica catchup.

Can it CashDB? Yes

Can it CashDB? Hell yeah, huge accord fan

Accord: Limitations

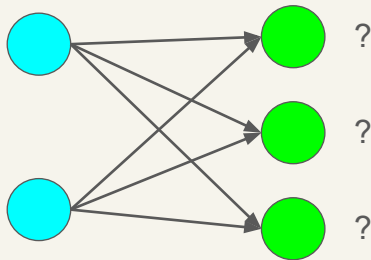
- Non-interactive / Stored procedure-only
- No long running transactions

CashDB would be only small OLTP transactions. It doesn't need interactivity. You would have to commit a debit/credit, and then execute it and find out if it was valid or not as like a durable OCC, but that's not the worst.

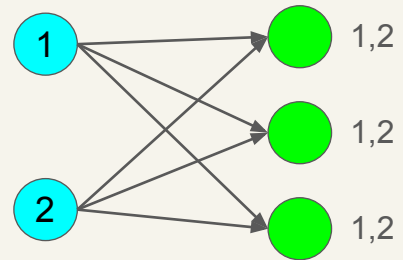
I like Accord a lot and think you should read about it.

Ordered Broadcast

Server Ordering



Client Ordering



And I wanted to take an extra quick moment to highlight that epaxos revisited ordered broadcast primitive.

The difficult part of leaderless consensus is agreeing upon ordering. Leaders are great at ordering!

Wouldn't it be great if everything could just be ordered upon arrival?

Client generated timestamps. Make up your order on your client, broadcast it with your message.

Variants of this idea show up in papers often.

EPaxos Revisited

Sarah Tollman
Stanford University

Seo Jin Park
MIT CSAIL

John Ousterhout
Stanford University

Abstract

This paper re-evaluates the performance of the EPaxos consensus protocol for geo-replication and proposes an enhancement that uses synchronized clocks to reduce operation latency. The benchmarking approach used for the original EPaxos evaluation does not trigger or measure the full impact of conflict behavior on system performance. Our re-evaluation confirms the original claim that EPaxos provides optimal median commit latency in a WAN, but it shows much worse tail latency than previously reported (more than 4x worse than Multi-Paxos). Furthermore, performance is highly sensitive to application workloads, particularly at the tail.

In addition, we show how synchronized clocks can be used to reduce conflicts in geo-replication. By imposing intentional delays on message processing, we can achieve roughly in-order deliveries to multiple replicas. When applied to EPaxos, this technique reduced conflicts by at least 50% without introducing additional overhead, decreasing mean latency by up to 7.5%.

1 Introduction

Consensus-based replication protocols such as Raft [23] and Multi-Paxos [15] play an important role in large-scale datacenter applications. These protocols have traditionally required two round-trip times (RTTs) between machines in order to

path. The original EPaxos evaluation included multiple scenarios for command interference, which plausibly measured best and worse case performance.

However, while replicating the EPaxos paper results, we discovered that the original benchmarking methodology did not trigger, and the metrics chosen did not capture, the full impact of conflicts on EPaxos performance. We reproduced the original EPaxos results, then broadened the evaluation to cover a wider range of potential conflict behavior. We also collected different metrics, which we believe more holistically reflect protocol performance.

Our re-evaluation draws different conclusions about EPaxos than the original evaluation. The original evaluation concluded that EPaxos has optimal commit latency in a WAN, where “optimal” is defined as 1 RTT to a simple majority of replicas. We agree that many operations can achieve this optimal latency, but we show that the percentage of operations completing in 1 RTT varies greatly depending on the workload. In one of our experiments, this percentage was as low as 65%. We also demonstrate that the latency increase for conflicting operations is substantial; even with a safe upper bound on execution latency many conflicting operations experience latency that is more than 4x worse than that of Multi-Paxos.

In the process of evaluating EPaxos more deeply, we discov-

What if you process messages only on or after their deadline?

Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks

[Technical Report]

Jinkun Geng*, Anirudh Sivaraman⁺, Balaji Prabhakar*, Mendel Rosenblum*

^{*}Stanford University, ⁺New York University

ABSTRACT

This paper presents a high-performance consensus protocol, Nezha, which can be deployed by cloud tenants without any support from their cloud provider. Nezha bridges the gap between protocols such as Multi-Paxos and Raft, which can be readily deployed and protocols such as NOPaxos and Speculative Paxos, that provide better performance, but require access to technologies such as programmable switches and in-network prioritization, which cloud tenants do not have.

Nezha uses a new multicast primitive called deadline-ordered multicast (DOM). DOM uses high-accuracy software clock synchronization to synchronize sender and receiver clocks. Senders tag messages with deadlines in synchronized time; receivers process messages in deadline order, on or after their deadline.

We compare Nezha with Multi-Paxos, Fast Paxos, Raft, a NOPaxos version we optimized for the cloud, and 2 recent protocols, Domino and TOQ-EPaxos, that use synchronized clocks. In throughput, Nezha outperforms all baselines by a median of 5.4 $\times$ (range: 1.9–20.9 $\times$). In latency, Nezha outperforms five baselines by a median of 2.3 $\times$ (range: 1.3–4.0 $\times$), with one exception: it sacrifices 33% latency compared with our optimized NOPaxos in one test. We also prototype two applications: a key-value store and a fair-access

Despite significant improvements in consensus protocols over the years, the status quo falls short in 2 ways. First, protocols such as Multi-Paxos [37] and Raft [58] can be (and are) widely deployed without help from the cloud provider. However, they only provide modest performance: latency in the millisecond range and throughput in the 10K requests/second range [17]. Second, high-performance alternatives such as NOPaxos [41], Speculative Paxos [63], NetChain [30], NetPaxos [30], and Mu [3], require technologies such as programmable switches, switch multicast, RDMA, priority scheduling, and control over routing—most of which are out of reach for the cloud tenant.¹

Here, we develop a protocol, Nezha, that provides high performance for cloud tenants without access to such technologies. Our starting point in designing Nezha is to observe that a common approach to improve consensus protocols is through *optimism*: in an optimistic protocol, there is a common-case fast path that provides low client latency, and there is a fallback slow path that suffers from a higher latency. Examples of this approach include Fast Paxos [36], EPaxos [56], Speculative Paxos [63], and NOPaxos [41].

For optimism to succeed, however, the fast path must indeed be the common case, i.e., the fraction of client requests that take the fast path should be high. For a sequence of client requests to

What if you also drop out-of-order messages?

Designing Distributed Systems Using Approximate Synchrony in Data Center Networks

Dan R. K. Ports Jialin Li Vincent Liu Naveen Kr. Sharma Arvind Krishnamurthy
University of Washington

{drkp, lij1, vincent, naveenks, arvind}@cs.washington.edu

Abstract

Distributed systems are traditionally designed independently from the underlying network, making worst-case assumptions (e.g., complete asynchrony) about its behavior. However, many of today's distributed applications are deployed in data centers, where the network is more reliable, predictable, and extensible. In these environments, it is possible to co-design distributed systems with their network layer, and doing so can offer substantial benefits.

This paper explores network-level mechanisms for providing *Mostly-Ordered Multicast (MOM)*: a best-effort ordering property for concurrent multicast operations. Using this primitive, we design *Speculative Paxos*, a state machine replication protocol that relies on the network to order requests in the normal case. This approach leads to substantial performance benefits: under realistic data center conditions, Speculative Paxos can provide 40% lower latency and $2.6\times$ higher throughput than the standard Paxos protocol. It offers lower latency than a latency-optimized protocol (Fast Paxos) with the same throughput as a throughput-optimized protocol (batching).

- Data center networks are more *extensible*. They are part of a single administrative domain. Combined with new flexibility provided by modern technologies like software-defined networking, this makes it possible to deploy new types of in-network processing or routing.

These differences have the potential to change the way distributed systems are designed. It is now possible to co-design distributed systems and the network they use, building systems that rely on stronger guarantees available in the network and deploying new network-level primitives that benefit higher layers.

In this paper, we explore the benefits of co-designing in the context of state machine replication—a performance-critical component at the heart of many critical data center services. Our approach is to treat the data center as an approximation of a synchronous network, in contrast to the asynchronous model of the Internet. We introduce two new mechanisms, a new network-level primitive called *Mostly-Ordered Multicast* and the *Speculative Paxos* replication protocol, which leverages approximate synchrony to provide higher performance in data centers.

What if the network provides best effort ordering?

Clairvoyant State Machine Replication

Rida Bazzi¹ and Maurice Herlihy²

¹ Arizona State University, Tempe, AZ, USA bazzi@asu.edu

² Brown University, Providence, RI, USA mph@cs.brown.edu

Abstract. We propose a new protocol for the generalized consensus problem in asynchronous systems subject to Byzantine server failures. The protocol solves the consensus problem in a setting in which information about conflict between transactions is available (such information can be in the form of transaction read and write sets). Unlike most prior proposals (for generalized or classical consensus), which use a leader to order transactions, this protocol is leaderless, and relies on non-skipping timestamps for transaction ordering. Being leaderless, the protocol does not need to pause for leader elections. The use of non-skipping times-

What if it's byzantine?

Building Consistent Transactions with Inconsistent Replication

Irene Zhang Naveen Kr. Sharma Adriana Szekeres
Arvind Krishnamurthy Dan R. K. Ports

University of Washington

{iyzhang, naveenks, aaasz, arvind, drkp}@cs.washington.edu

Abstract

Application programmers increasingly prefer distributed storage systems with strong consistency and distributed transactions (e.g., Google's Spanner) for their strong guarantees and ease of use. Unfortunately, existing transactional storage systems are expensive to use – in part because they require costly replication protocols, like Paxos, for fault tolerance. In this paper, we present a new approach that makes transactional storage systems more affordable: we eliminate consistency from the replication protocol while still providing distributed transactions with strong consistency to applications.

We present TAPIR – the Transactional Application Protocol for Inconsistent Replication – the first transaction protocol to use a novel replication protocol, called inconsistent replication, that provides fault tolerance without consistency. By enforcing strong consistency only in the transaction protocol, TAPIR can commit transactions in a single round-trip and or-

trend, notably Google's Spanner system [13], which guarantees linearizable transaction ordering.¹

For application programmers, distributed transactional storage with strong consistency comes at a price. These systems commonly use replication for fault-tolerance, and replication protocols with strong consistency, like Paxos, impose a high performance cost, while more efficient, weak consistency protocols fail to provide strong system guarantees.

Significant prior work has addressed improving the performance of transactional storage systems – including systems that optimize for read-only transactions [4, 13], more restrictive transaction models [2, 14, 22], or weaker consistency guarantees [3, 33, 42]. However, none of these systems have addressed both latency *and* throughput for general-purpose, replicated, read-write transactions with strong consistency.

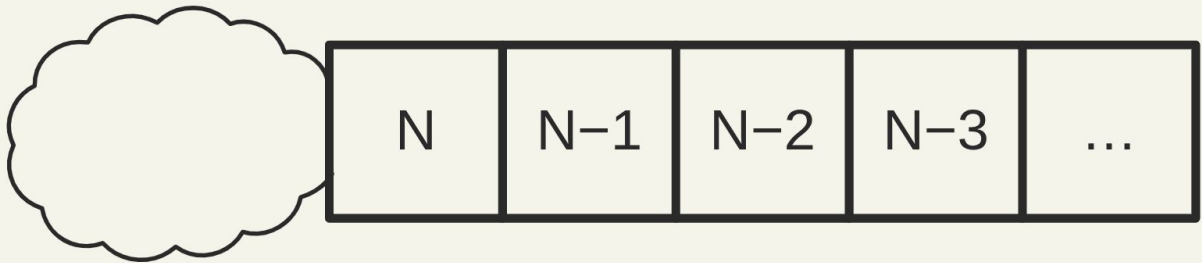
In this paper, we use a new approach to reduce the cost of replicated, read-write transactions and make transactional

What if it's a transaction timestamp?

No details for time

and we're ending on TAPIR so I can mention System #3 that I *think* SurrealDB is putting TAPIR into production which is pretty cool

Async Leader Tricks



Our last bag of tricks are a few very related ideas because applying any one of them, gives you the same high level architecture.

There's a cloud of unordered commutative operations from which you pull a log out of afterwards with some great-at-ordering leadered consensus.

You can do a 1RTT quorum write into the unordered pool, and remember that Linearizable Quorum Reads says you can do a 1RTT read from a quorum of non-leaders, so you're left only stalling read-after-writes.

Exploiting Commutativity For Practical Fast Replication

Seo Jin Park
Stanford University

John Ousterhout
Stanford University

Abstract

Traditional approaches to replication require client requests to be ordered before making them durable by copying them to replicas. As a result, clients must wait for two round-trip times (RTTs) before updates complete. In this paper, we show that this entanglement of ordering and durability is unnecessary for strong consistency. The Consistent Unordered Replication Protocol (CURP) allows clients to replicate requests that have not yet been ordered, as long as they are commutative. This strategy allows most operations to complete in 1 RTT (the same as an unreplicated system). We implemented CURP in the Redis and RAMCloud storage systems. In RAMCloud, CURP improved write latency by $\sim 2\times$ ($14\ \mu\text{s} \rightarrow 7.1\ \mu\text{s}$) and write throughput by $4\times$. Compared to unreplicated RAMCloud, CURP's latency overhead for 3-way replication is just $1\ \mu\text{s}$ ($6.1\ \mu\text{s}$ vs $7.1\ \mu\text{s}$). CURP transformed a non-durable Redis cache into a consistent and durable storage system with only a small performance overhead.

1 Introduction

Fault-tolerant systems rely on replication to mask individual failures. To ensure that an operation is durable, it cannot

Consistent Unordered Replication Protocol (CURP) reduces the overhead for replication by taking advantage of the fact that most operations are commutative, so their order of execution doesn't matter. CURP supplements a system's existing replication mechanism with a lightweight form of replication without ordering based on *witnesses*. A client replicates each operation to one or more witnesses in parallel with sending the request to the primary server; the primary can then execute the operation and return to the client without waiting for normal replication, which happens asynchronously. This allows operations to complete in 1 RTT, as long as all witnessed-but-not-yet-replicated operations are commutative. Non-commutative operations still require 2 RTTs. If the primary crashes, information from witnesses is combined with that from the normal replicas to re-create a consistent server state.

CURP can be easily applied to most existing systems using primary-backup replication. Changes required by CURP are not intrusive, and it works with any kind of backup mechanism (e.g. state machine replication [31], file writes to network replicated drives [1], or scattered replication [26]). This is important since most high-performance systems optimize their backup mechanisms, and we don't want to lose

CURP is a really nice introduction to this idea, and hides a description of how to apply it to raft in the appendix

Jetpack: Consensus Made Generally Fast

Ze Tang¹, Zihao Zhang^{1,†}, Weihai Shen¹, Jicheng Shi², Shuai Mu¹

¹*Stony Brook University* ²*DatenLord*

Abstract

Classic consensus protocols such as Raft require 2 round-trip times (RTTs) for a client to commit a command. Recent protocols reduce this latency to 1 RTT through a carefully designed fast path, but these fast paths are tightly coupled to their host protocols and cannot be retrofitted to existing systems. This paper presents Jetpack, a plugin framework that adds a 1-RTT fast path to existing consensus protocols with minimal modifications.

Jetpack issues commands concurrently to both a fast path and the original path, committing when the fast path succeeds while ensuring the original path honors the same decision. The key challenge is maintaining this guarantee across view changes: we identify the *view change hazard*, a subtle correctness issue where promises made during stable operation can become invalid after leader elections. Jetpack captures this in two structural requirements and two design principles that a fast-path layer can apply across consensus environments, conditions that are easy to overlook when a fast-path idea is

to a supermajority. Subsequent systems such as EPaxos [49], Tapir [65], and SwiftPaxos [59] realize this 1-RTT fast path in complete system designs.

However, the fast paths in these systems are tightly coupled to their protocol designs. They are neither intended nor designed to be generalized to other systems. One cannot extract the fast path from these protocols and apply it to existing systems without adopting the entire design. Yet replacing the consensus layer of a mature system is practically impossible. Even new systems tend to choose reliable, widely adopted protocols. For example, most recent consensus systems [38, 61, 68, 7, 2, 12] in industry are Raft-based.

Given this gap between state-of-the-art and practice, this paper asks: can we develop a universal fast-path framework applicable to a broad range of consensus protocols and systems? We answer this question with Jetpack, a plugin approach that layers fast-path optimization on top of existing protocols without requiring extensive protocol redesign or code modification.

It was then later generalized to work on more consensus algorithms and fixed in Jetpack

“Commutative”

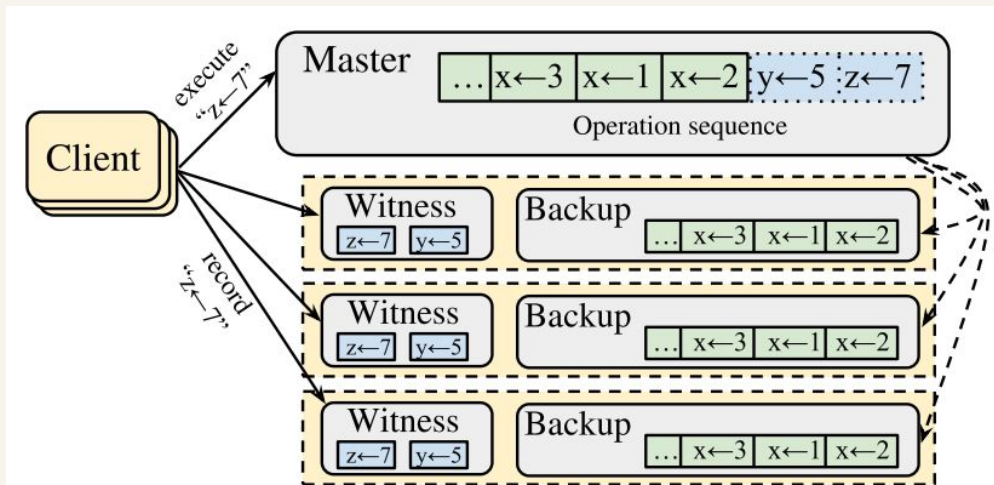


Figure 2: CURP architecture for $f=3$ fault tolerance.

Both stop as defining “commutative” operations as those that affect two different keys. So pulling this diagram from CURP, they accepted a quorum write to Y and to Z, but another write to Y would be forced to stall while the leader works on propagating Y into the log.

CURP almost got a production implementation in CNCF Xline, but they went straight from Beta to Archived.

Exploiting Nil-Externality for Fast Replicated Storage

Aishwarya Ganesan
VMware Research

Andrea C. Arpaci-Dusseau
University of Wisconsin – Madison

Ramnathan Alagappan
VMware Research

Remzi H. Arpaci-Dusseau
University of Wisconsin – Madison

Abstract

Do some storage *interfaces* enable higher performance than others? Can one identify and exploit such interfaces to realize high performance in storage systems? This paper answers these questions in the affirmative by identifying *nil-externality*, a property of storage interfaces. A nil-externalizing (nilext) interface may modify state within a storage system but does not externalize its effects or system state immediately to the outside world. As a result, a storage system can apply nilext operations lazily, improving performance.

In this paper, we take advantage of nilext interfaces to build high-performance replicated storage. We implement SKYROS, a nilext-aware replication protocol that offers high performance by deferring ordering and executing operations until their effects are externalized. We show that exploiting nil-externality offers significant benefit: for many work-

may modify state within a storage system but does not externalize its effects or system state immediately to the outside world (apart from the acknowledgment itself). As a result, a storage system can apply a nilext operation in a deferred manner after acknowledgment, improving performance.

In this paper, we exploit nil-externality to design high-performance replicated storage that offers strong consistency (i.e., linearizability [36]). A standard approach today to building such a system is to use a consensus protocol like Paxos [44], Viewstamped Replication (VR) [52], or Raft [62]. For example, Facebook's ZippyDB uses Paxos to replicate RocksDB [73]; Harp builds a replicated file system using VR [53]; other examples exist as well [7, 17, 18, 22].

A storage system built using this standard approach performs several actions before it returns a response to a request. Roughly, the system makes the request durable (if it is an

To take these ideas one step further, you need the Skyros paper introducing Nil-Externality

Nil-Externality

- Operations that have an effect but return nothing.

System	Update		Read
	Nilext	Non-nilext	
RocksDB	put,write, delete,merge		get,multiget
LevelDB	put,write, delete		get,multiget
Memcached	set	add ^e ,delete ^e ,cas ^r ,replace ^e , append ^e ,decr ^r ,incr ^r ,prepend ^e	get,gets

Table 1. Nil-externality in Storage Systems. *The table shows which operations are nilext in popular key-value systems. I^e denotes that update interface I is non-nilext because it returns an execution error (e.g., key not found); I^r denotes a non-nilext update that returns an execution result.*

Nil-externality is a generalization of blind writes to all operations that don't return any state about the system.

Skyros lets you just keep quorum writing nil-external operations to the same key.

Each replica remembers its local order, and a leadered consensus can pull back out a correct linearizable order later.

Nil-Externality has no `_direct_` implementation in industry, but we've been doing similar logic on the concurrency control side by not checking conflicts between blind writes in serializable MVCC databases for a long time.

Skyros tacks on commutative operations like increments but doesn't discuss it much.

MDCC: Multi-Data Center Consistency

Tim Kraska Gene Pang Michael J. Franklin Samuel Madden*

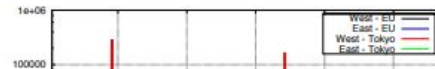
UC Berkeley *MIT
{kraska, gpang, franklin}@cs.berkeley.edu madden@csail.mit.edu

ABSTRACT

Replicating data across multiple data centers not only allows moving the data closer to the user and, thus, reduces latency for applications, but also increases the availability in the event of a data center failure. Therefore, it is not surprising that companies like Google, Yahoo, and Netflix already replicate user data across geographically different regions.

However, replication across data centers is expensive. Inter-data center network delays are in the hundreds of milliseconds and vary significantly. Synchronous wide-area replication is therefore considered to be unfeasible with strong consistency and current solutions either settle for asynchronous replication which implies the risk of losing data in the event of failures, restrict consistency to small partitions, or give up consistency entirely. With MDCC (Multi-Data Center Consistency), we describe the first optimistic commit protocol, that does not require a master or partitioning, and is strongly consistent at a cost similar to eventually consistent protocols. MDCC can commit transactions in a single round-trip across data centers in the normal operational case. We further pro-

significantly as shown in Figure 1 for message delays between different Amazon regions. Traditional commit protocols for ensuring transactional consistency in distributed databases (e.g., two-phase commit (2PC)) were not designed for these and highly variable latencies that occur in wide area networks. First, existing protocols are pessimistic and are required to prepare all resources involved in a transaction by essentially acquiring a lock on the resource, causing an additional message delay and blocking the resource for other concurrent transactions. These locks are typically held for at least two network round trip times, which, as shown in Figure 1, can often result in several hundred milliseconds additional latency. Such additional latency can significantly impact the usability of websites; for example [29] shows that an additional 200 milliseconds of latency can result in a significant drop in user satisfaction and “abandonment” of websites.



I'll give MDCC credit as the other paper I know of which directly took on commutative updates under a leaderless generalized paxos model

The Escrow Transactional Method

PATRICK E. O'NEIL

Computer Corporation of America

A method is presented for permitting record updates by long-lived transactions without forbidding simultaneous access by other users to records modified. Earlier methods presented separately by Gawlick and Reuter are comparable but concentrate on "hot-spot" situations, where even short transactions cannot lock frequently accessed fields without causing bottlenecks. The Escrow Method offered here is designed to support nonblocking record updates by transactions that are "long lived" and thus require long periods to complete. Recoverability of intermediate results prior to commit thus becomes a design goal, so that updates as of a given time can be guaranteed against memory or media failure while still retaining the prerogative to abort. This guarantee basically completes phase one of a two-phase commit, and several advantages result: (1) As with Gawlick's and Reuter's methods, high-concurrency items in the database will not act as a bottleneck; (2) transaction commit of different updates can be performed asynchronously, allowing natural distributed transactions; indeed, distributed transactions in the presence of delayed messages or occasional line disconnection become feasible in a way that we argue will tie up minimal resources for the purpose intended; and (3) it becomes natural to allow for human interaction in the middle of a transaction without loss of concurrent access or any special difficulty for the application programmer. The Escrow Method, like Gawlick's Fast Path and Reuter's Method, requires the database system to be an "expert" about the type of transactional updates performed, most commonly updates involving incremental changes to aggregate quantities. However, the Escrow Method is extendable to other types of updates.

and 1986 Escrow transactions let you extend that commutative logic to ... almost commutative operations.

Escrow Transactions

- Assume $X=10$, constraint $X>0$
- Track value if all increments abort, all decrements succeed

INCR(X, 5)		Worst case: $X=10$
	DECR(X,5)	Worst case: $X=5$
COMMIT	COMMIT	

Escrow says: let increments and decrements commute as long as it's impossible for your constraint to be violated. If in the absolute worst case where all the increments abort, and all decrements succeed, then the value still remains above 0, then what do you care just treat them all as commutative.

Escrow has only one implementation I'm aware of, Oracle 23c introduced it under the name "lockfree reservations", but, Oracle is not a quorum database

High Throughput Escrow Algorithms for Replicated Databases*

(Extended Abstract)

Narayanan Krishnakumar Arthur J. Bernstein
Dept. of Computer Science, SUNY at Stony Brook,
Stony Brook, NY 11794-4400.
email : {nkris,art}@sbcs.sunysb.edu
tel : (516) 632-8434

Abstract

The traditional correctness criterion in replicated databases is one-copy serializability. However, this criterion is sometimes restrictive and degrades performance. Recent research has therefore focused on utilizing application semantics to increase transaction throughput in certain high-performance applications. One such application involves resource allocation. To improve concurrency in such a system, the transaction escrow (TE) and site escrow (SE) algorithms have been proposed. In this paper, we present a generalized site escrow algorithm (GSE) that provides high site autonomy and throughput. GSE requires only a loose synchronization between sites, and employs the mechanisms of quorum locking and background gossip messages. We perform a comparison between GSE and TE, and outline regions in which GSE performs

circumstances so that the benefits of both algorithms can be utilized. Finally, we present a variant of GSE that does not use locking.

Keywords : replication, resource allocation, replica control, escrow.

1 Introduction

Distributed databases can exhibit better performance than centralized databases since transactions can execute concurrently at different sites. However, data may then have to be accessed at a remote site, and this can result in message passing delays. Replicating the data reduces these delays: when a transaction wishes to read the data, an expensive remote access is not required. Replication also increases the availability of the data, since the data can be accessed even if a

To apply this to quorum systems, you need the 1992 paper: High Throughput Escrow Algorithms for Replicated Databases which tells you that in the standard 3 of 5 quorum setup, the worst case value needs to stay above 40% of the current account value instead of 0. It's not a terribly hard fix.

Can it CashDB? Yes

So if we squish all these ideas together... can it cashdb?

Rich Operations are Good Operations

- Account Transfers? Escrow
- Log appends? Nil-external
- Rich operations make everything better
 - Consensus over Logs: pipelining
 - Consensus over Graphs: decreases conflicts

CashDB only supports an escrow-able account transfer operation and a nil-external log append. I obviously designed CashDB for this argument.

In isolation this async leader trick is not 100% leaderless, but it's a great step towards better latency.

But I'm also mentioning it because leveraging blind and commutative operations makes EVERY OTHER ALGORITHM BETTER and we never do it.

In Consensus over Logs, Nil-Externality and Escrow would help guide us as to what operations are safe to pipeline with a leaderless log.

In Consensus over Graphs, it means we get to reduce the dependencies tracked, and thus end up on the fast path more often.

Increments are the best operation ever because you can treat them as commutative OR a nil-external fused together read-modify-write that returns nothing, and we still basically never special case them?!? We just execute them under exclusive locks, it saddens me.

END

- There are cool new papers and you should read them.
- Recent quorum systems have gotten much better.
- We can still build even better quorum systems.
- We choose replication via imitation.

Alright, I'm out of time and out of content, so:

we have very recently gotten much better algorithms for leaderless replication
we have very recently gotten much better production-or-almost-there examples of
how to write good quorum databases
there are still more good ideas we have yet to apply to their full extent

I do not think we will see a proliferation of quorum databases until we have good
production examples to copy from.

Please Implement A Quorum Database

So I implore you. Choose something different. Choose something with better latency. Choose something that lets you drop a "why quorums are better" blog post every time your competitor has an outage.

Please implement a quorum database.

Thank you.