Cloud Consensus Protocols with Optimistic Connectivity

Guanzhou Hu
josehu.com

Dissertation Defense
Computer Sciences
University of Wisconsin – Madison
June 30, 2025

Committee:

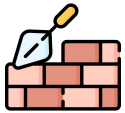
- Andrea Arpaci-Dusseau (advisor)
- Remzi Arpaci-Dusseau (advisor)
- Michael Swift
- Tej Chajed
- Xiangyao Yu
- Kassem Fawaz (ECE)

Overview

Study, design, and implement better **Consensus** protocols and systems …

… for modern replicated **Cloud** services …

… using our proposed principle of **Optimistic Connectivity** …



… and other underpinning contributions to consensus research.

Overview

Consensus protocols

enable

strongly-consistent
highly-available
State Machine Replication

which is at the heart of

fault-tolerant cloud services

Modern cloud imposes
"4D" challenges

- Density
- Distance
- Diversity
- Dynamism

Existing consensus solutions did not consider

We present
two protocols

Crossword

Bodega

that follow the principle of
Optimistic Connectivity

to tackle composite "4D" challenges
and advance the state-of-the-art in
cloud consensus systems

Overview: Crossword

Density + Diversity + Dynamism

Problem: cloud consensus systems face a dynamic mix of small and large payloads

Approach: adaptive erasure-coded consensus with a shard count – quorum size tradeoff

- for large requests, optimistically choose larger quorum sizes for reduced data transfer
- upon failures, smoothly switch to conservative quorum size configurations
- configuration is tunable per consensus instance, giving adaptivity
- off-the-critical-path gossiping between followers to retain graceful leader failover

Result: over 2x performance of existing protocols, 1.32x throughput in CockroachDB!

Overview: Bodega

Distance + Diversity + Dynamism

Problem: wide-area consensus delivers slow reads, leaving location affinity untapped

Approach: adaptive roster composition of responder nodes that serve reads locally

- per-key tunable selection of responder roles according to client locations & workloads
- novel roster leases algorithm to enable fault-tolerant updates to rosters, always retaining linearizability and availability with negligible overhead (embedded in heartbeats)
- optimizations: optimistic holding, early accept notifications, smart roster coverage, …

Result: performance on par with sequentially-consistent etcd & ZooKeeper deployments!

Overview

Consensus protocols

enable

strongly-consistent
highly-available
State Machine Replication

which is at the heart of

fault-tolerant cloud services

Modern cloud imposes
"4D" challenges

- Density
- Distance
- Diversity
- Dynamism

We present
two protocols

Crossword

Bodega

that follow the principle of
Optimistic Connectivity

Summerset
KV Testbed

Unified Consistency Levels
Hierarchy & Checker

Formal but Practical TLA+
Specifications

Outline

Problem Context	Main Contributions	System Implementation	Supportive Contributions	Discussions & Future Work
Consensus & SMR	*Optimistic Connectivity*	KV Testbed: **Summerset**	TLA+ Specifications	Related Work Areas
Classic Protocols	Protocol: **Crossword**	Linearizability Checking	Consistency Models Unified	Future Work
Cloud-Era Challenges	Protocol: **Bodega**		Jepsen Checker	

Outline

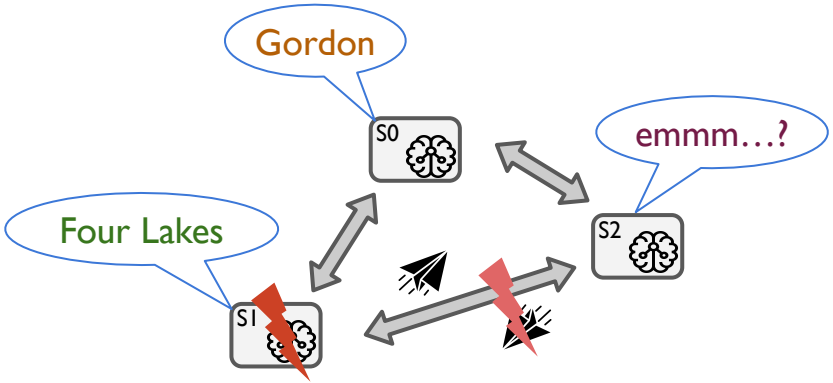
Problem Context	Main Contributions	System Implementation	Supportive Contributions	Discussions & Future Work
Consensus & SMR	*Optimistic Connectivity*	KV Testbed: **Summerset**	TLA+ Specifications	Related Work Areas
Classic Protocols	Protocol: **Crossword**	Linearizability Checking	Consistency Models Unified	Future Work
Cloud-Era Challenges	Protocol: **Bodega**		Jepsen Checker	

Consensus & State Machine Replication

The Consensus Problem

Consensus := reaching agreement among message-passing processes despite failures

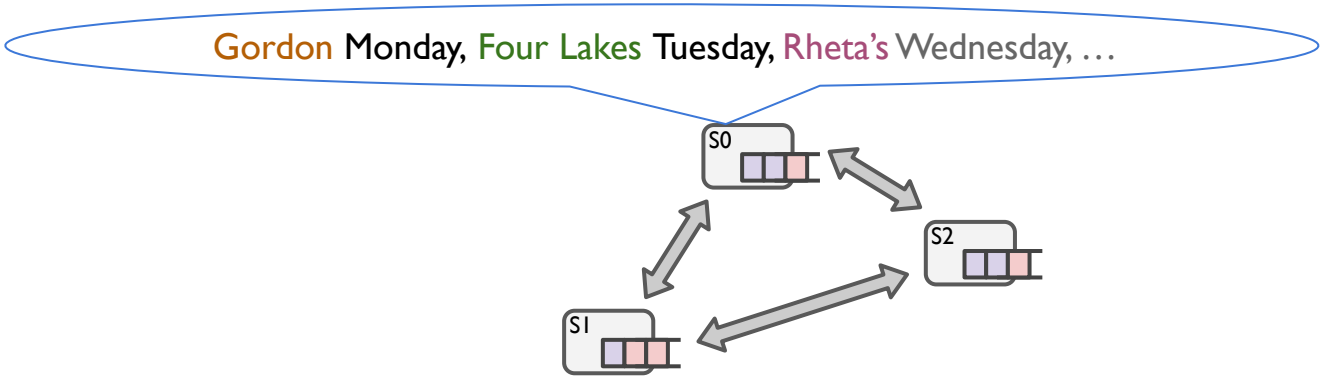
"single-decree"



Goal: everyone knows where to meet for dinner
(which one doesn't matter)

Multi-decree Consensus

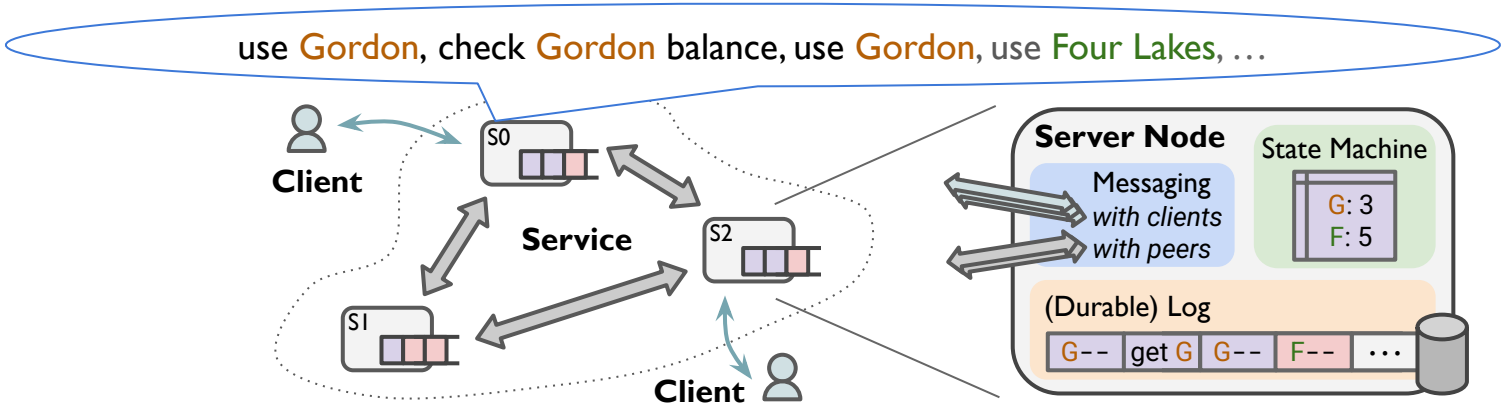
Multi-decree consensus := reaching a continual sequence of agreements



Goal: everyone knows where to meet for dinner for every night onwards

State Machine Replication (SMR)

SMR := multi-decree consensus where the sequence is a log of state machine commands to be executed in the same order on all nodes



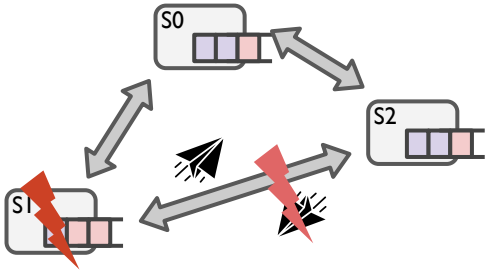
Goal: a fault-tolerant service that tracks meal plan balance, replicated 3 ways

Failure Model

What failures do we handle?

Fail-stop node:

- may *crash* or respond arbitrarily *slowly* at any time
- may *recover* at any time



Asynchronous network:

- message may be *lost*, *duplicated*, or delivered arbitrarily *slowly*
- messages may be delivered *out-of-order*
- network partitioning is covered

What failures do we not handle?

Byzantine failures: malicious nodes, malicious/corrupted messages

Consistency & Availability Requirement

Consistency: how "correct" is correct replication?

- All replicas agree on a single serial order  (exception: commutative commands)
- If op1 finished earlier than op2 started, op1 must precede op2 in the order

Linearizability => as if an atomic single-node service

Availability: how many concurrent faults to tolerate before blocking progress?

- Measure: counting # of failed nodes, including
  - fail-stopped nodes
  - nodes that have trouble sending out messages
- Goal: match the availability level of classic consensus protocols

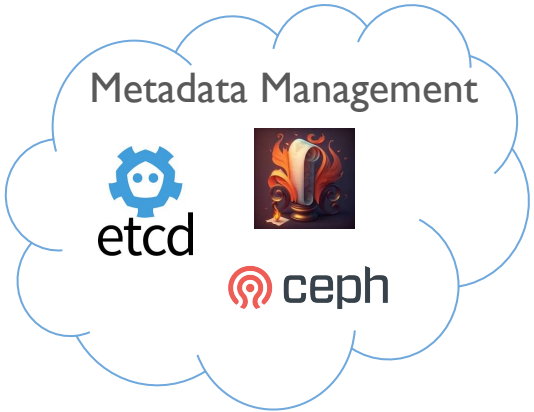
Classic Consensus Protocols

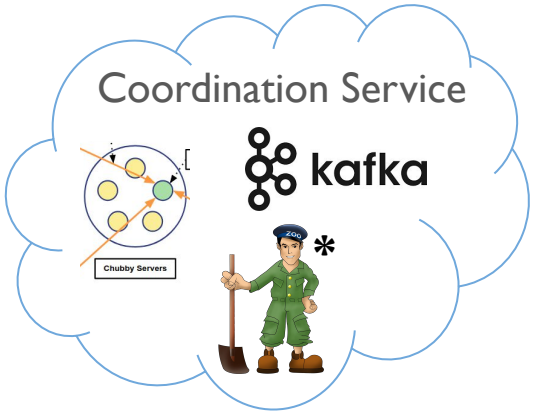
Consensus in the Wild

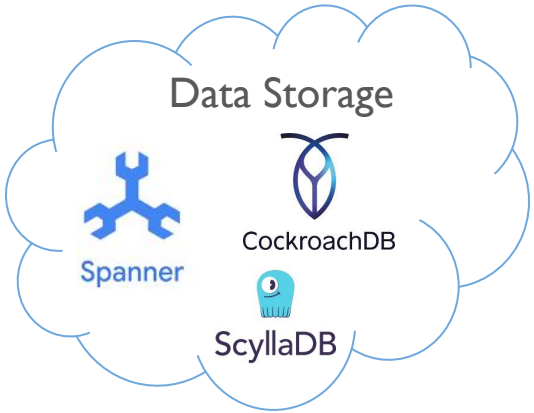
Classic protocols: MultiPaxos & variants [15], Viewstamped Replication [16], Raft [17]

Inherent duality between MultiPaxos and latter ones [2]

Consensus-infused systems across the cloud landscape:



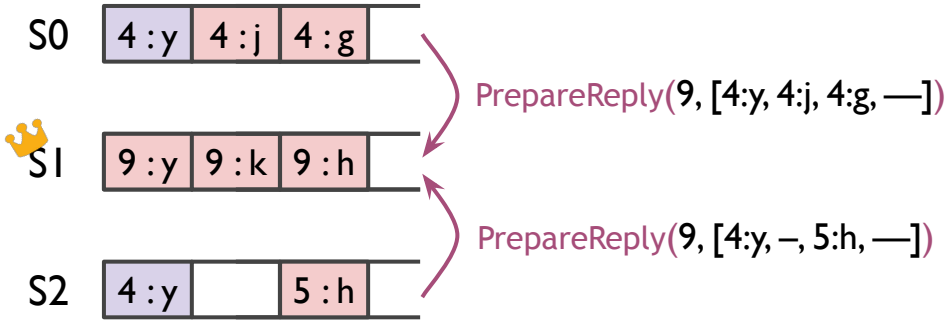




Classic Protocol: MultiPaxos

Prepare Phase – some node S steps up as leader and gathers must-know history

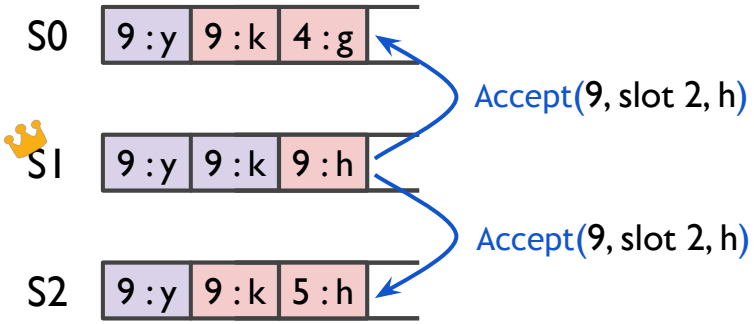
- S chooses a higher-than-seen, unique *ballot* number *b*
- S broadcasts Prepare(*b*)
- Receiver replies with a "covering-all" PrepareReply(*b*, [b'_0:v'_0, b'_1:v'_1, …]) containing the highest ballot it has ever accepted for each slot and its value
- S, upon getting >= majority replies, is effectively "elected" as leader; for each non-committed slot in order:
  - if no values found among replies:
    - can try any value, so wait for client input
  - otherwise:
    - immediately do Accept Phase using the value with the highest ballot among replies



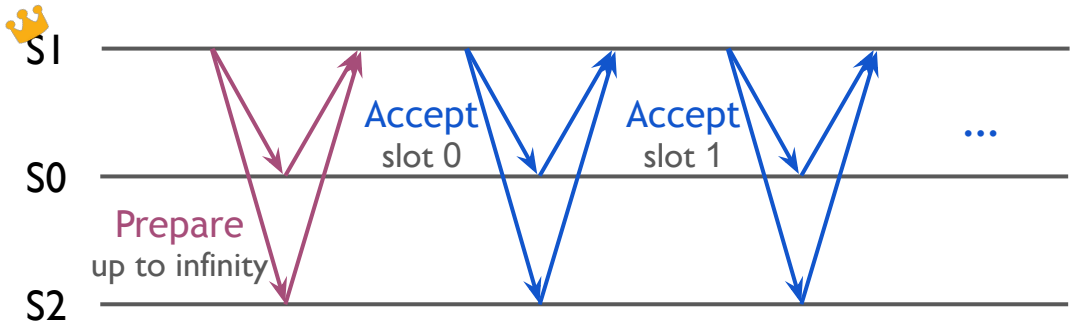
Classic Protocol: MultiPaxos

Accept Phase per slot – leader S establishes agreement of a slot with followers

- S checks the next slot that's pending acceptance, and chooses safe value *v* for it according to the last slide
- S broadcasts Accept(*b, slot, v*)
- Receiver checks if ballot *b* >= the largest ballot it has ever seen?
  - if yes, accept the value:
    reply with AcceptReply(*b, slot*)
  - otherwise, ignore
- S, upon getting >= majority replies, *commits* the slot and tells followers asynchronously; the contiguously committed prefix of the log can be scheduled for *execution* on the SM



MultiPaxos Timeline View



Only the Accept round is needed in failure-free cases without competing leaders

Cloud-Era Challenges: What Changed?

The "4D" Challenges

Density

Payload Size Heaviness

Distance

Geo-Scale Distribution

Diversity

Diverse Workloads & Heterogeneous Hardware

Dynamism

Constant Changes over Time

Density

Density

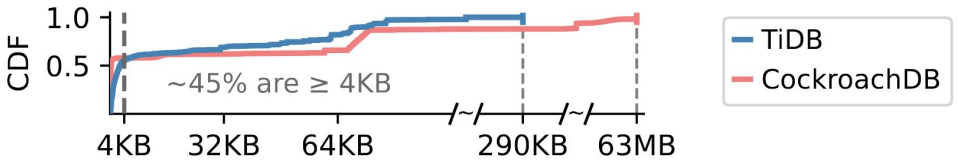
Payload Size Heaviness

Metadata operations can become MBs in size as reported [3]

PaceMaker: When ZooKeeper Arteries Get Clogged in Storm Clusters

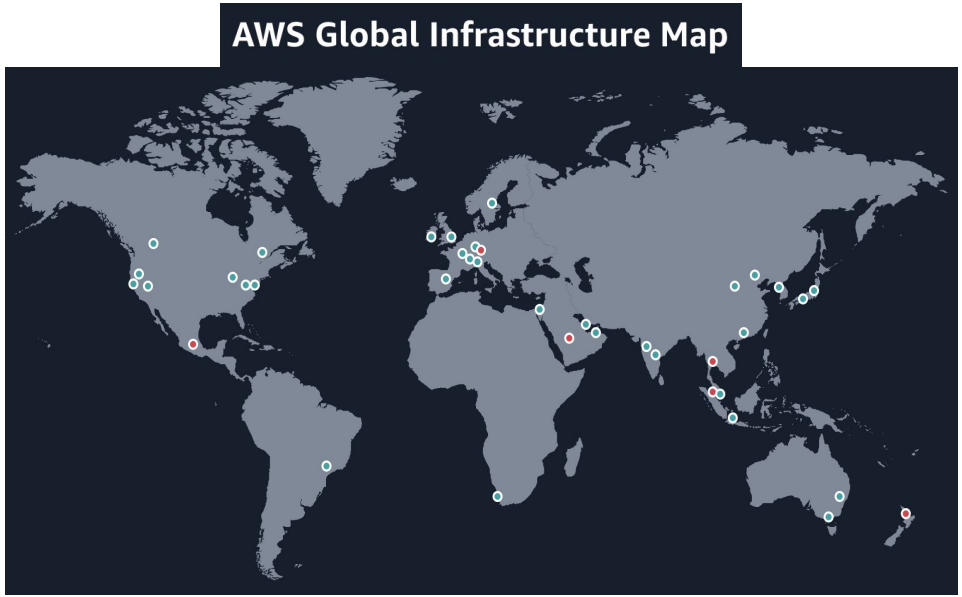
Sanket Chintapalli, Derek Dagit, Robert Evans, Reza Farivar, Zhuo Liu
Kyle Nusbaum, Kishorkumar Patil, Boyang Peng
Yahoo Inc.
{schintap, derekd, evans, rfarivar, zhuol, knusbaum, kpatil, jerrypeng}@yahoo-inc.com

Raft replication payload size CDF
profiled from 200 warehouses TPC-C



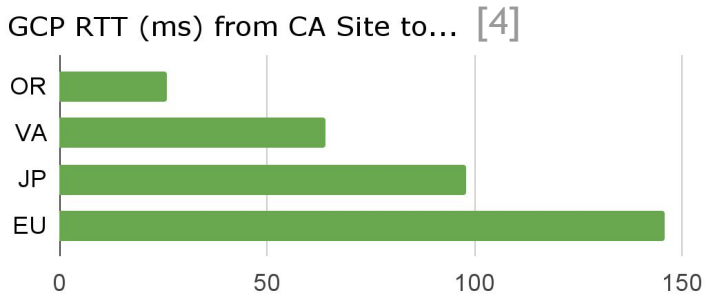
Distance

Clients can be globally distributed too!



Distance

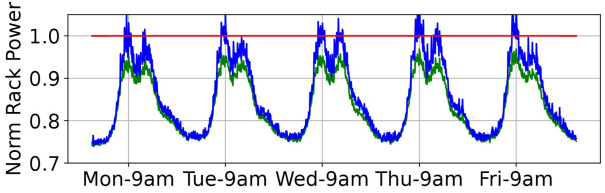
Geo-Scale Distribution



Diversity & Dynamism

Workload:
- read/write mix
- object affinity
- location affinity
- rate variance

Hardware:
- network
- storage
- memory
- compute



Power draw of an Azure rack over time [5]

Workload patterns

Failures and recovery

Hardware conditions

Diversity

Diverse Workloads & Heterogeneous Hardware

Dynamism

Constant Changes over Time

Existing Consensus Didn't Consider "4D"

Existing consensus protocols

rarely express Density and Distance in their designs

offer no runtime adaptability to Diversity and Dynamism

Crossword and **Bodega** solve two concrete manifestation of "4D"

Optimistic Connectivity design principle

Outline

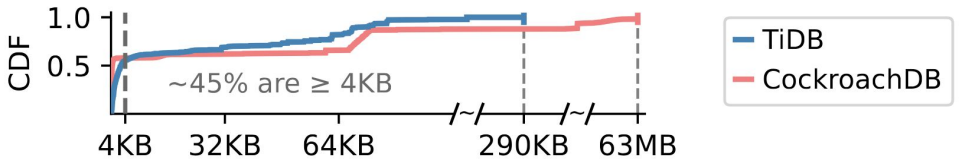
Problem Context	Main Contributions	System Implementation	Supportive Contributions	Discussions & Future Work
Consensus & SMR	*Optimistic Connectivity*	KV Testbed: **Summerset**	TLA+ Specifications	Related Work Areas
Classic Protocols	Protocol: **Crossword**	Linearizability Checking	Consistency Models Unified	Future Work
Cloud-Era Challenges	Protocol: **Bodega**		Jepsen Checker	

Crossword: Adaptive Consensus for Dynamic Data-Heavy Workloads

Problem: Dynamic Data-Heavy Workloads

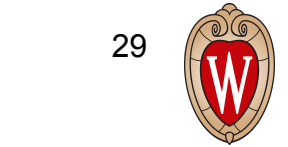
Density + Diversity + Dynamism

Raft replication payload size CDF
profiled from 200 warehouses TPC-C



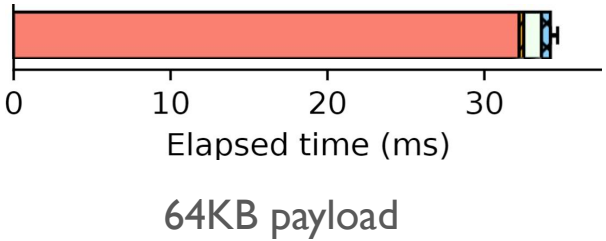
Consensus modules in cloud systems face a dynamic mix of small and large requests

- Cloud databases: CockroachDB, TiDB, ScyllaDB, F1/Spanner, …
- Object storage: Gaios, Amazon S3, Dynamo, …
- Metadata of large-scale systems: Colossus/GFS, Apache Storm, …
- Additional factors: request batching, fluctuations in hardware environment



Not in the Design Equation of Classic Protocols

MultiPaxos, Raft, … replicate the **full** command in entirety onto replicas



- Leader→follower Accept msg
- Writing to durable WAL
- Follower→leader AcceptReply (**majority** quorum)
- Commit & execution

In a cluster of $n = 2f + 1$ nodes,
tolerates f faults in every attempt

Payload Reduction? Erasure Coding

Reed-Solomon Code

:= an RS codeword

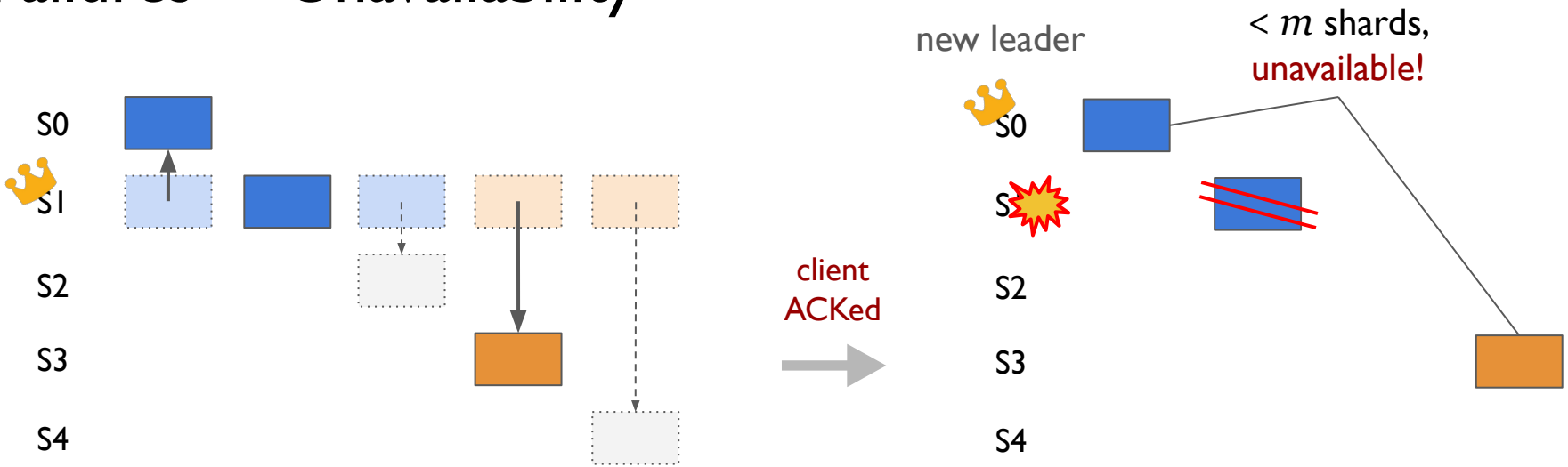
d data shards p parity shards

Can recover the original data from any d shards

RSPaxos [9] and CRaft [10] replicate only **one shard** onto each replica

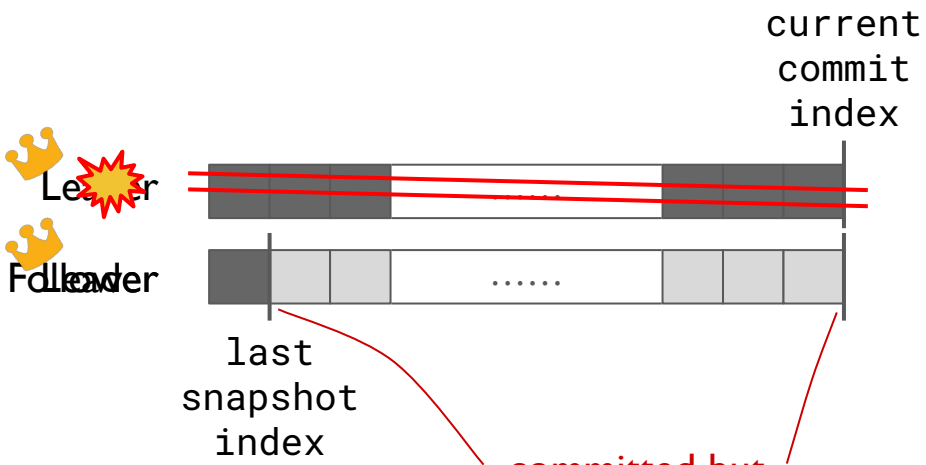
- Assuming using a configuration of $d = m$ = majority size, $p = n\text{-}m$
- Critical-path data transfer time reduced to $1/m$
  - RS code computation time is negligible

Failures → Unavailability



- RSPaxos uses fixed quorum size of 4 and tolerates a single failure
- CRaft introduces a (slow) fallback mechanism to full-replication, but still vulnerable to any concurrent failures during the long fallback job

Leader Failover → Intense Reconstruction Traffic



Insufficiencies of Previous Work

- Reduced fault-tolerance level
  - Not drop-in replacement to classic consensus protocols
  - Tolerates 1 failure with a 5-way replication
- Ungraceful leader failover behavior
  - Followers do not see complete commands, lagging infinitely behind the leader
  - Significant reconstruction traffic after leader failover
- Rigid shard assignment scheme
  - Always uses disjoint shard assignment, always optimizing for large payloads
  - No adaptability with delay-optimized configurations under true dynamic workloads

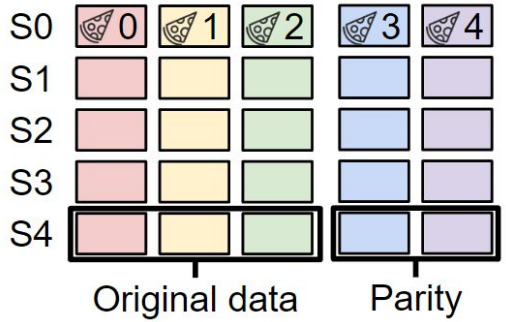
Goals for Crossword

- Integrate RS coding to improve performance for data-heavy workloads
- Same fault-tolerance level as classic consensus protocols ($f = \lfloor n/2 \rfloor$)
- Flexibility and adaptability to workload sizes and environment changes
- Graceful leader failover behavior without long pauses

RS Codeword Space

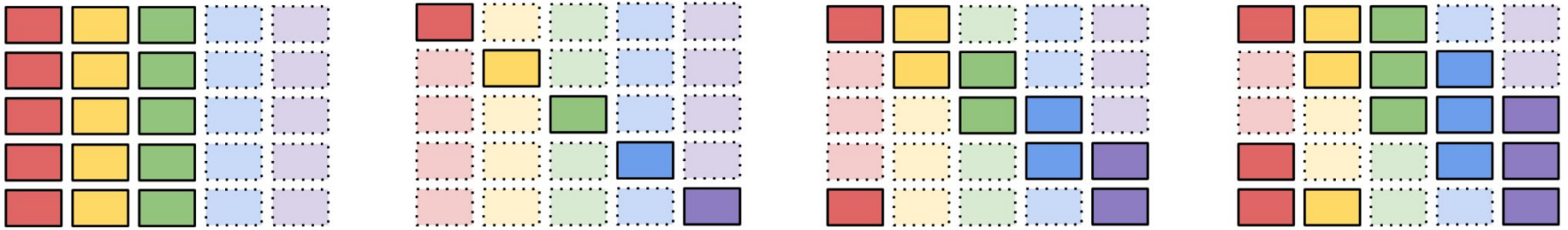
Observation: mappings from RS shards to servers need not be disjoint!

Introduce a new per-instance notation, RS codeword space :=



Opens up a new dimension in protocol design: what shard(s) to assign to which server(s)

Shard Assignment Policies



(a) MultiPaxos & Raft

(b) RSPaxos & CRaft, also Balanced RR, $c = 1$

(c) CROSSWORD, Balanced RR, $c = 2$

(d) CROSSWORD, Balanced RR, $c = 3$

"Balanced Round-Robin (RR) assignment policies"

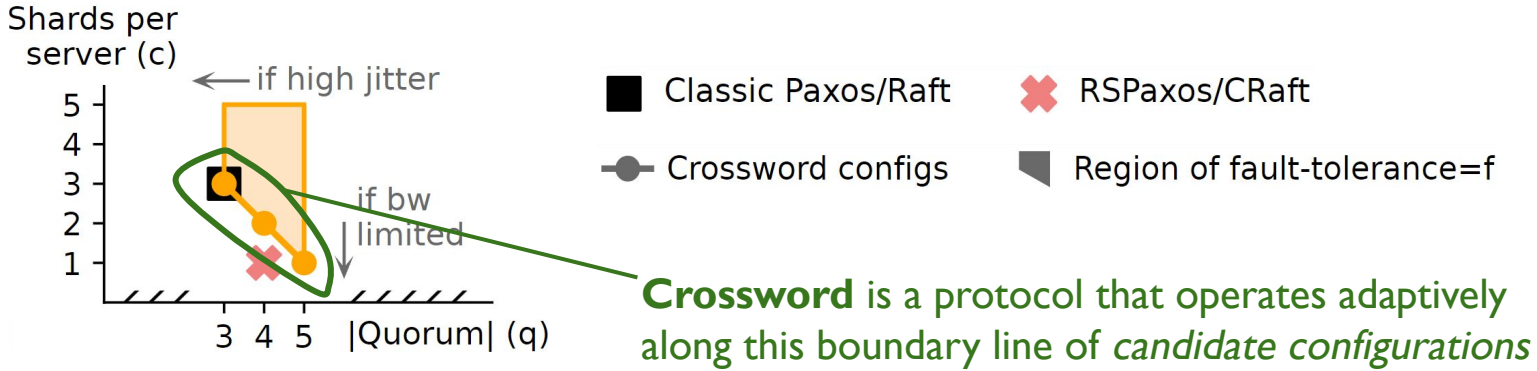
Assign shards $[i, i+c)$ to server i, rounding back

Availability Constraint Boundary

There is a **trade-off** between #shards assigned per server (c) and the quorum size (q)

The correct constraint boundary that **retains fault-tolerance** $= f$ is:

$$q + c \geq n + 1$$



Crossword is a protocol that operates adaptively along this boundary line of *candidate configurations*

Choosing the Best Configuration

Leader maintains a real-time updated simple linear regression model per follower s

$$t_s(v) = d_s + 1/b_s * v$$

delay bandwidth payload size

For each instance:

chooses the (c, q) pair that minimizes estimated completion time of a q quorum

more sophisticated methods possible

Crossword Protocol in a Nutshell

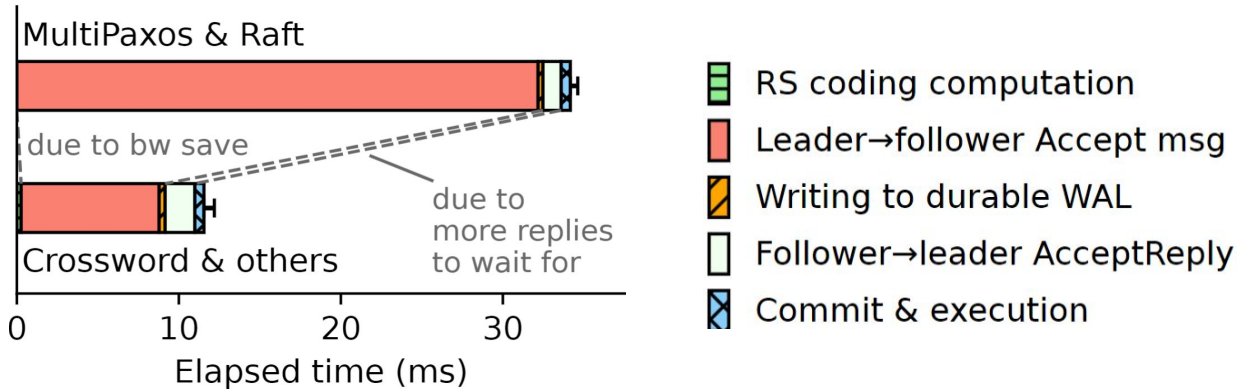
Leader

1. Upon receiving a payload:
   1.1. Choose the best configuration
   1.2. Broadcast Accept messages to followers, each carrying (a subset of) shards
2. Upon receiving AcceptReply ← follower:
   2.1. Check if safe quorum size reached
   2.2. Mark entry committed; notify followers
3. Execute committed entry, reply to clients

Follower

1. Upon receiving Accept message ← leader:
   1.1. Store the carried shard(s) durably
   1.2. Send back AcceptReply
2. Upon being noticed about new commit:
   2.1. Mark entry committed
   2.2. (cannot execute right away)

Instance Performance Breakdown



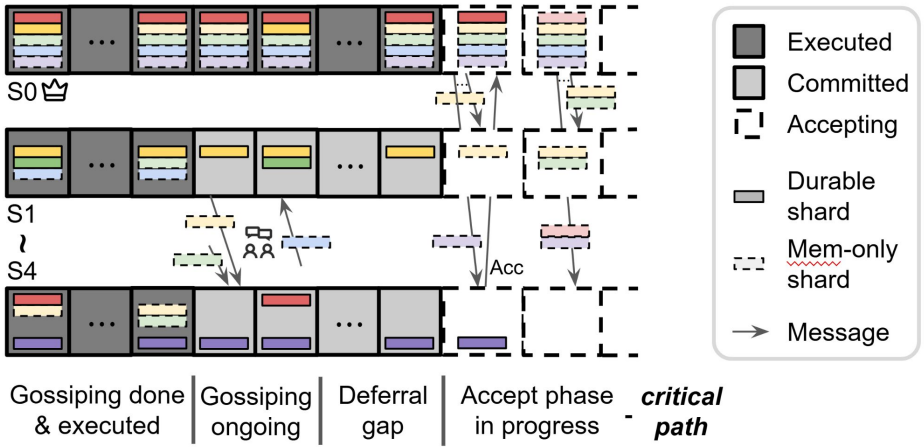
64KB payload, symmetrical delay case

What about the leader failover problem?

Follower Gossiping (details omitted in this talk)

Let followers **gossip** with each other **in the background** about their missing shards

- Keeps them **(almost) up-to-date** with committed data → graceful failover
- Happens **asynchronously** → minimal impact to critical path, can be delayed
- Uses an **Round Robin pattern** amongst followers → amortized traffic
- Introduces a ***deferral gap*** to prevent unnecessary queries to latest entries



Crossword Protocol in a Nutshell

Leader

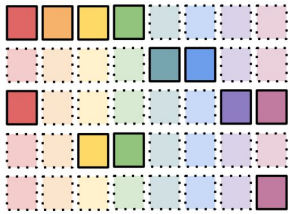
1. Upon receiving a payload:
   1.1. Choose the best configuration
   1.2. Broadcast Accept messages to followers, each carrying (a subset of) shards
2. Upon receiving AcceptReply ← follower:
   2.1. Check if safe quorum size reached
   2.2. Mark entry committed; notify followers
3. Execute committed entry, reply to clients

Follower

1. Upon receiving Accept message ← leader:
   1.1. Store the carried shard(s) durably
   1.2. Send back AcceptReply
2. Upon being noticed about new commit:
   2.1. Mark entry committed
3. Periodically, trigger follower gossiping for committed instances (except a deferral gap at log tail), fetch shards from adjacent followers
   3.1. When payload fully known, execute

Left-out Details

- Modifications to the Prepare phase & failover protocol
- Optimizations to follower gossiping: deferral gap, gossip batching, …
- Support for *unbalanced* assignment policies with general-form constraints
- Log storage space saving
- Different cluster sizes, full protocol diagram



(e) CROSSWORD, Unbalanced case

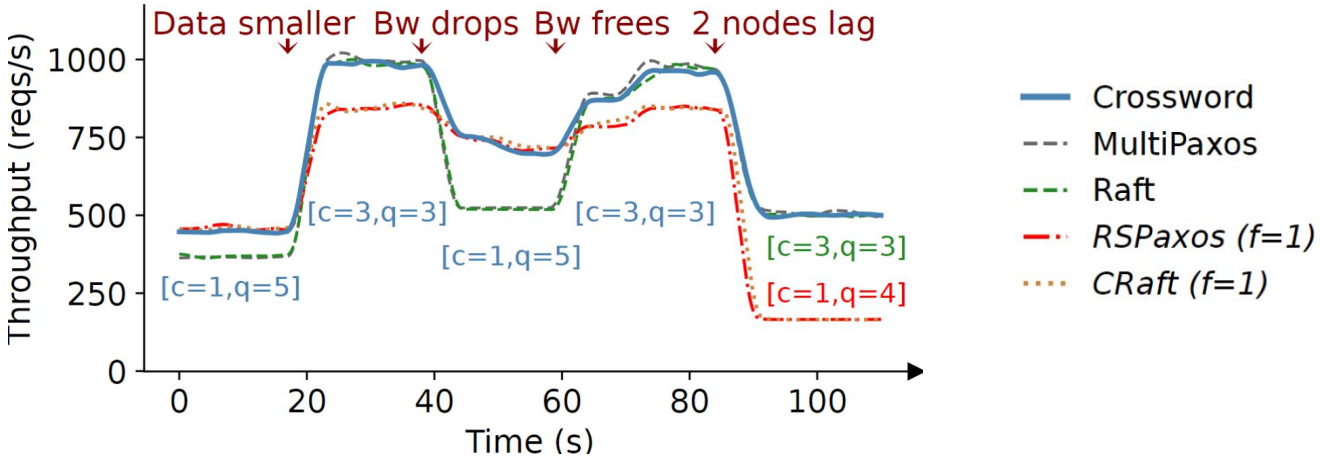
Evaluation Setup

Implemented on Summerset, evaluated on CloudLab `c220g2` machines

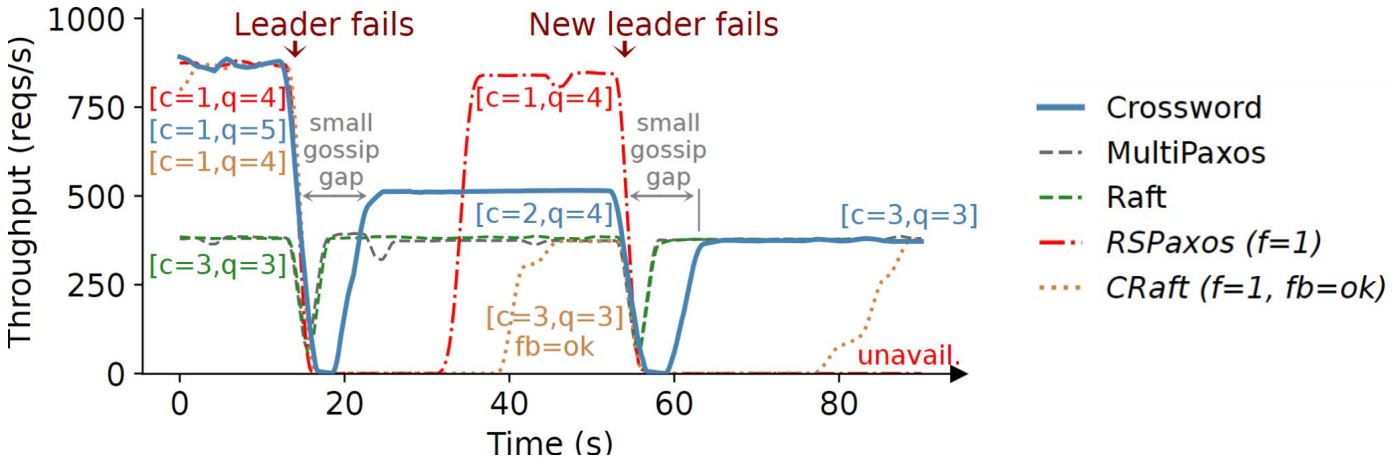
- mainly 5 nodes
- 40 CPU threads, 160GB DRAM, S3500 SATA SSD, X520 NIC
- Node-node network bandwidth 1Gbps, delay ~2ms
- Launch 1 server process per machine to form a cluster
- Run closed-loop clients distributed across all machines

Critical Path Performance

Adaptability to Dynamism



Leader Failover Behavior



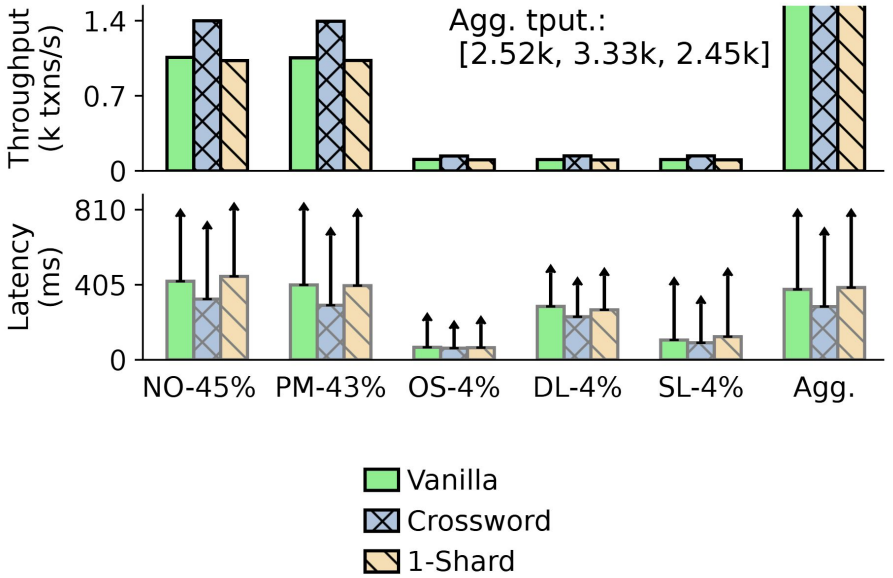
TPC-C over CockroachDB

Raft module integration

- implemented a Crossword prototype in the Raft replication module of CockroachDB
- uses 4KB / 8KB thresholds as configuration heuristics

TPC-C benchmark

- 200 warehouses
- 400 concurrent workers



Optimistic Connectivity: A Design Principle

Patterns Observed

Classic protocols are "pessimistic" about failures

- always uses a quorum size that guards against f failures in every attempt

Larger quorum sizes can be good for performance in some cases

- in Crossword's scenario, enables fewer critical-path data transfer
- but may not always be the best choice
- and may affect the fault tolerance of the protocol

Optimistic Connectivity

Optimistic Connectivity  :=

be *optimistic* that a large quorum size *configuration* can be established if advantageous

BUT, reserve the ability to switch to *conservative configurations* upon timeout

Crossword as concrete example:

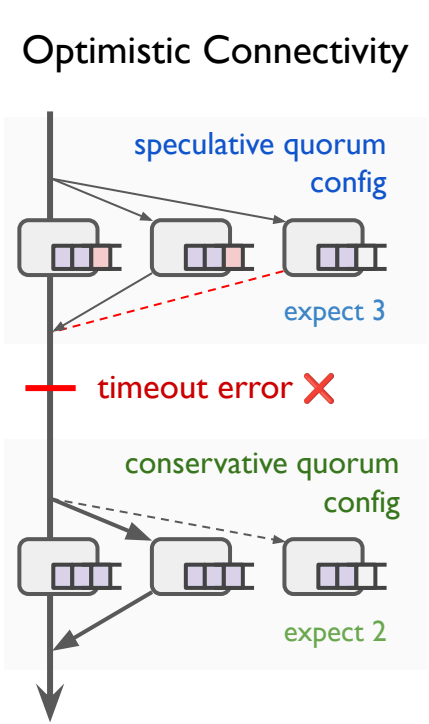
use a (larger q, smaller c) configuration when payload size is large w.r.t. available bandwidth

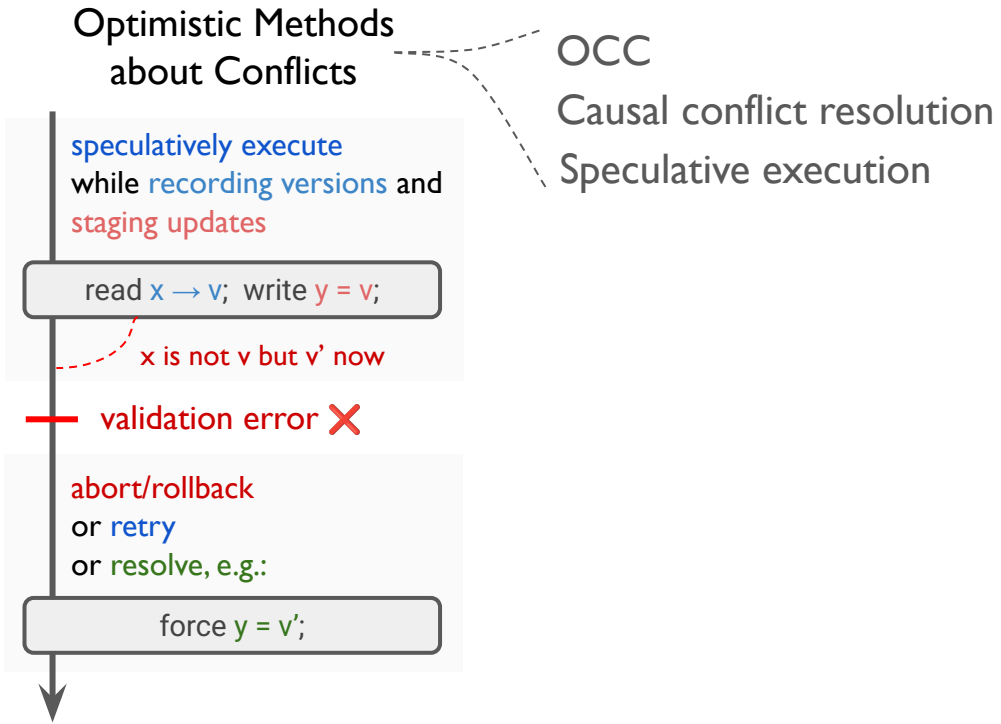
BUT, always able to redo with a (smaller q, larger c) configuration upon failures

Effect:

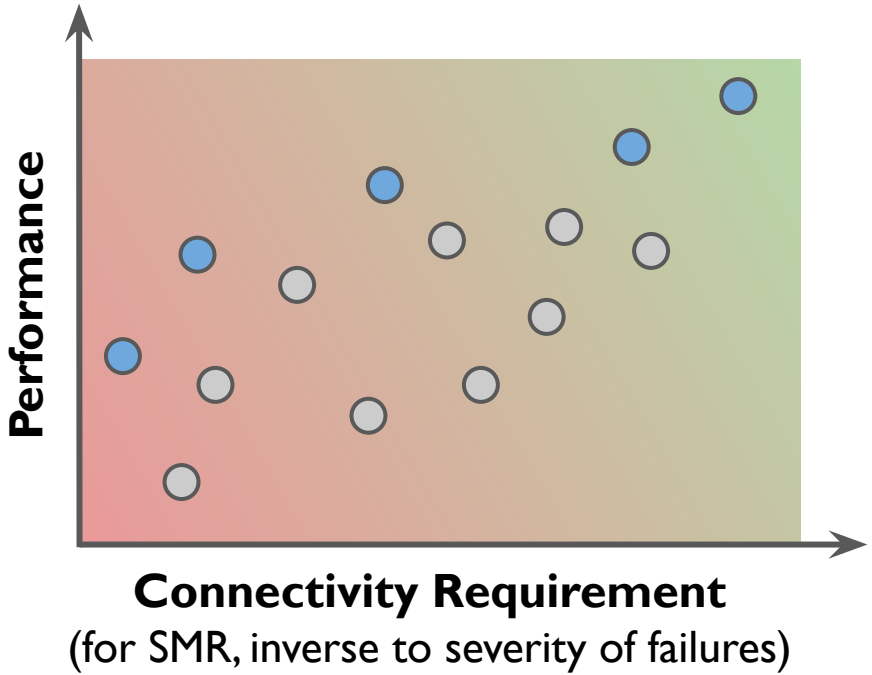
- turns a rigid protocol into an adaptive protocol
- squeezes out performance without losing correctness and availability

Analogy with Optimistic Methods about Conflicts





Configuration Space



Bodega: Wide-Area Consensus with Always-Local Linearizable Reads

Problem: Geo-Scale Linearizable Reads

Distance + Diversity + Dynamism

Wide-area (geo-scale) consensus performs poorly

- Writes cannot avoid wide-area latency
- Can we exploit location affinity for Reads?

Easily > 500ms latency

Not in the Design Equation of Classic Protocols

MultiPaxos, Raft, … process all requests using **leader**-initiated **majority** quorums

slow: client ↔ leader ↔ majority

Leaderless protocols EPaxos [11], PQR [12], … allow **near-client** quorums

still slow: client ↔ (super-)majority

Atypical quorums Dynamic quorums [13], FPaxos [14], … allow **asymmetric** quorum sizes

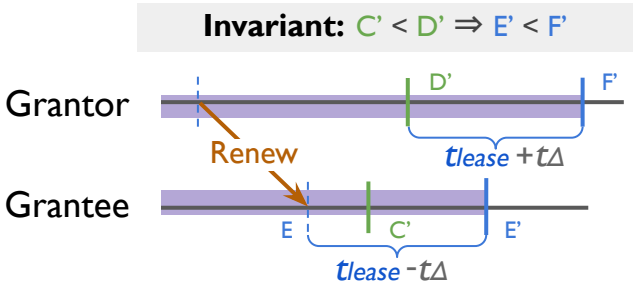
still slow: statically configured (write qsize, read qsize) pair

Local Read? Need Leases

We want a replica to serve linearizable reads from nearby clients locally,
without compromising fault tolerance for writes

=> Requires a "promise" mechanism that is aware of time

Lease := a limited-time, refreshable promise that a grantor makes to a grantee



- ensures grantor always holds the promise longer than any grantee
- assumes bounded clock drift between the two (common in modern cloud hardware)

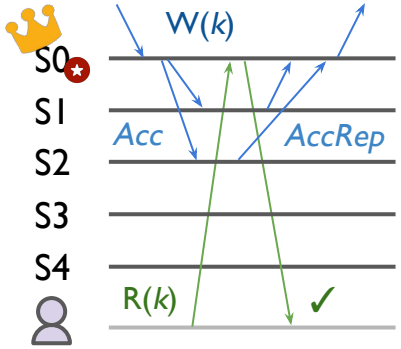
Leader Leases – Local at Leader Only

How can leases help empower local reads?

Leader Leases: establish **stable leadership** so the leader can safely serve reads locally

- if node S holds >= majority# of leases, it knows no other node is acting as leader
- therefore, S can serve read requests directly with the latest committed write value

promise (periodically) not to step up as competing leader or vote for other node



only local at leader, not at the nearest follower at will

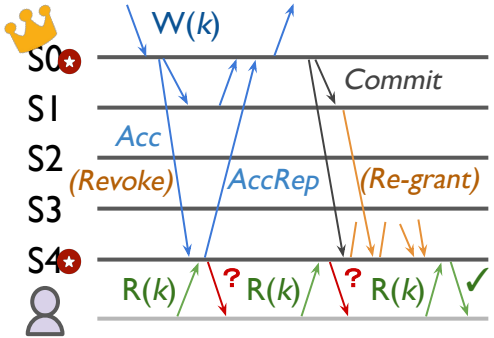
Quorum Leases – Write Interference (details omitted in this talk)

Applying the Leader Leases idea to a follower is non-trivial

Quorum Leases: grant leases to a follower to allow local reads **in the absence of writes**

- if node S holds >= majority# of leases on key *k*, AND its latest known value for *k* is in committed status, it knows no newer value could have been committed for *k*
- therefore, S can serve read requests arriving at it directly with that value

promise (periodically) not to accept any write to key *k*, unless actively revoked



local read interrupted during any writes to lease-covered keys

Goals for Bodega

- Utilize leases to enable local linearizable reads
- Same fault-tolerance level as classic consensus protocols ($f = \lfloor n/2 \rfloor$)
- Local read anywhere: at arbitrary replicas
- Local read at any time: minimal interference from writes
- Configurable assignment of reader replicas for flexible key ranges

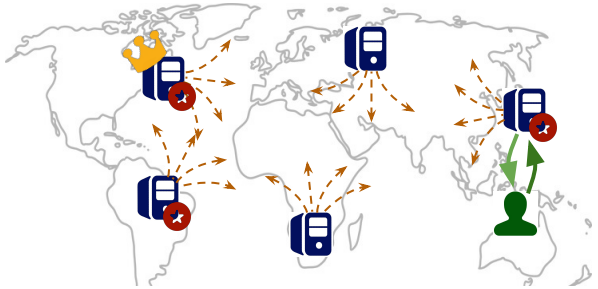
Roster: A Generalization of Leadership

Observation: leases can protect richer cluster metadata than leadership

Leadership := who's the stable leader → **Roster** := { who's the stable leader, and for each key, who are *responders* to serve local reads

Leader Leases: all-to-one pattern → **Roster Leases**: all-to-all pattern

leader holding >= m means can local read



a responder holding >= m means can local read

Optimistic Connectivity

Bodega is a protocol that uses the roster to achieve *optimistic quorum composition*

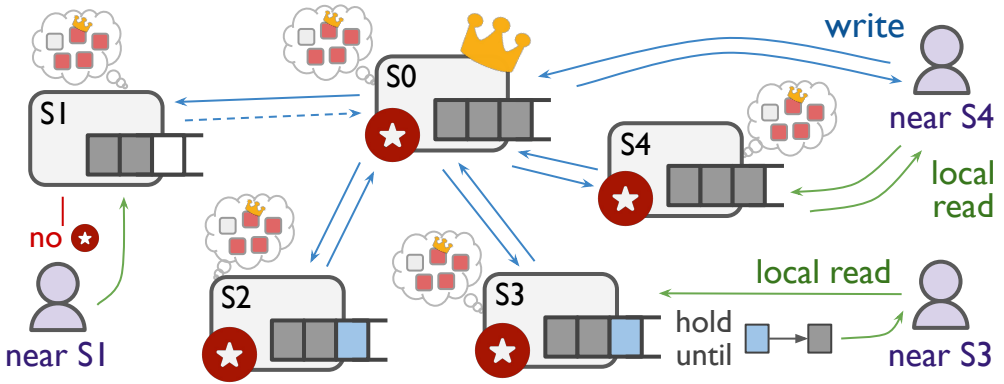
- in normal case, expect all the responders of a key are reachable by its writes
  in return for local read capability at these responders (1-node read "quorum"s)
- upon failures, leases allow a safe update to the roster, removing unresponsive nodes
- roster can also be updated proactively to adapt to workload changes

Norcal Case Operations

Write

Read, assume lease count >= m

- local read successful
- local read with **holding**
- not a responder



latest value not committed yet:

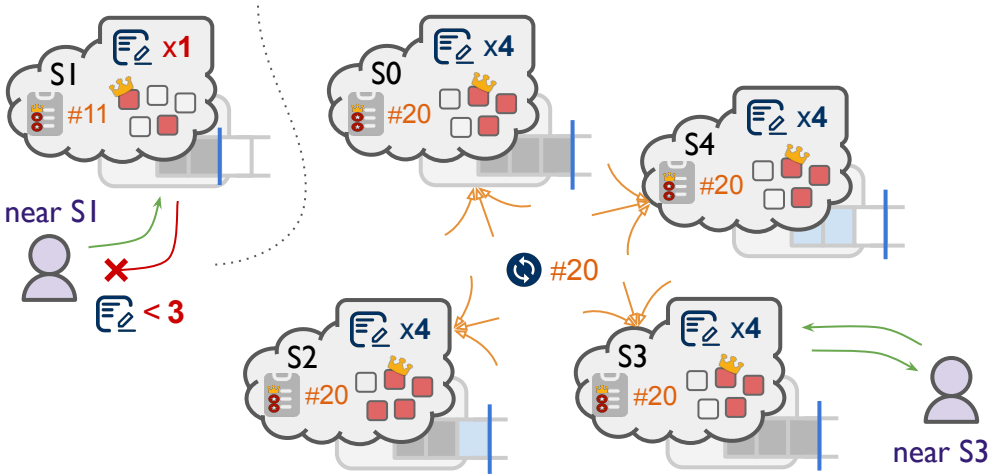
hold the request until commit notification received from leader

Roster Leases Embedded in Heartbeats

Every unique ballot# corresponds to a roster

Lease renewals are piggybacked onto existing peer-to-peer heartbeats

- virtually zero overhead



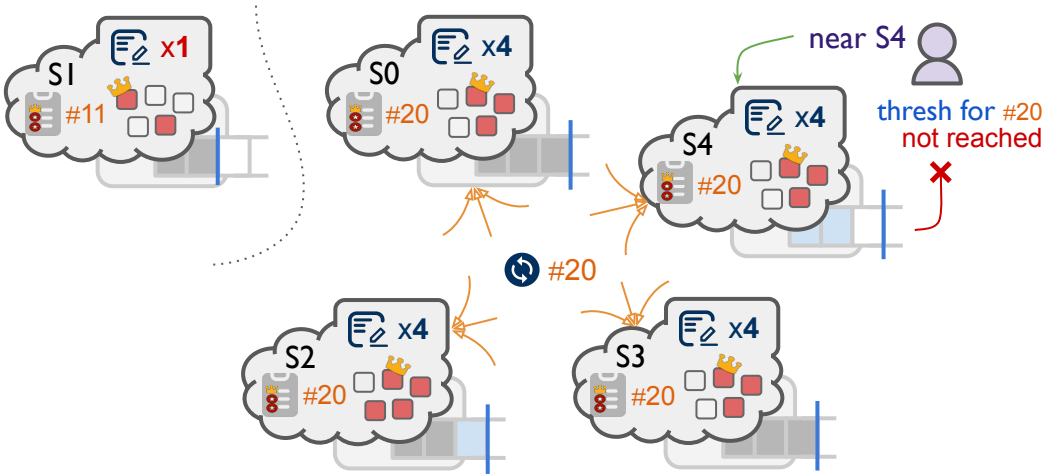
One Little Detail: Safety Threshold (details omitted in this talk)

Holding >= m leases might not be enough

- consider a lagged-behind node becoming a responder

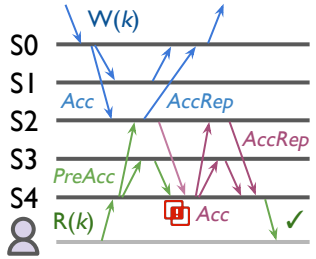
Corner case: when transitioning to a new roster, cannot start serving local reads until knowing all previously majority-accepted slots

- communicated within the first round of lease messages, again no overhead

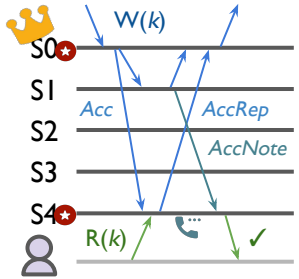


Left-out Details

- Full comparison with **related work**
  - Leaderless approaches: Mencius, EPaxos, SwiftPaxos, PQR
  - Read leases: Megastore, Quorum Leases
  - With external configuration oracle: Hermes, Pando

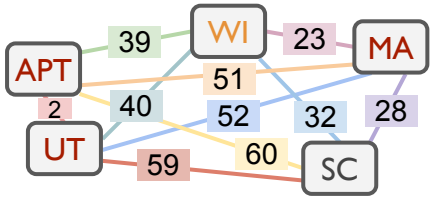


- Optimizations:
  - Optimistic holding
  - Early accept notifications
  - Smart roster changes
  - Lightweight heartbeats



Evaluation Setup

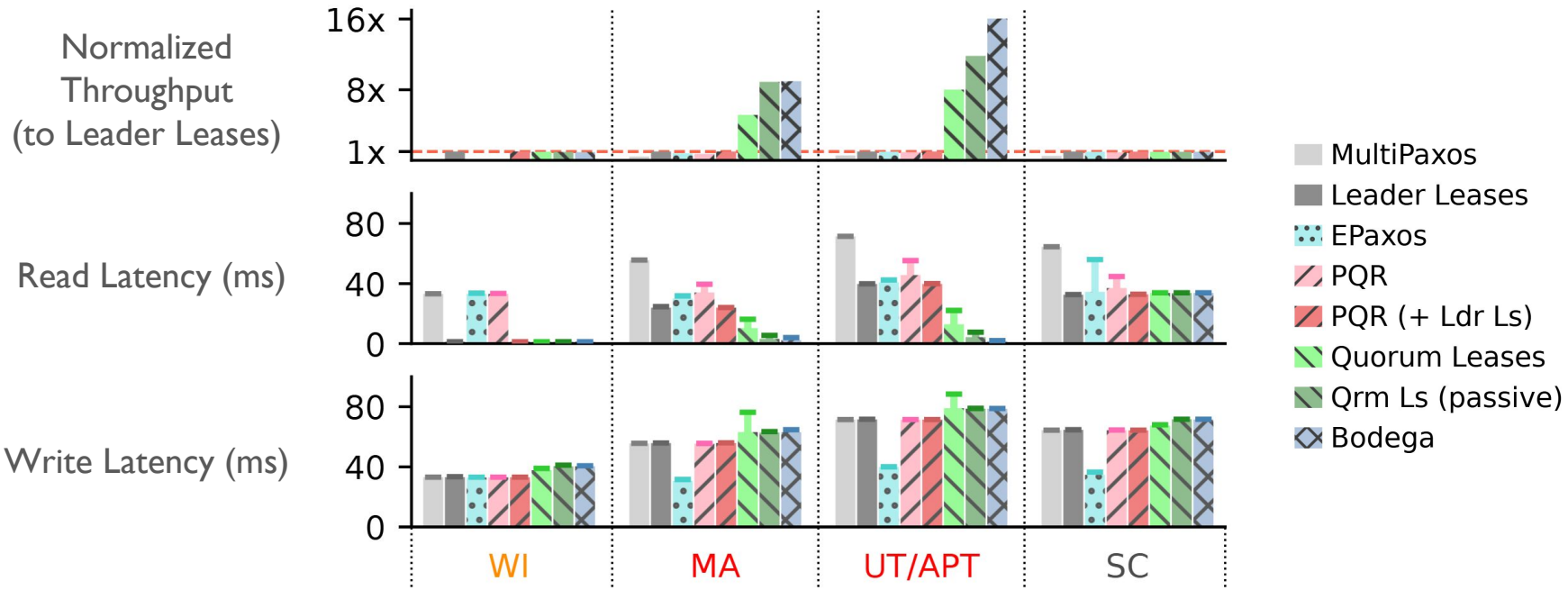
Implemented on **Summerset**, evaluated across 5 CloudLab sites



- mainly 5 nodes
- WI-`c220g5`, UT-`xl170`, SC-`c6320`, MA-`rs620`, APT-`r320`
- Launch 1 server process per machine to form a cluster
- Run closed-loop clients distributed across all machines

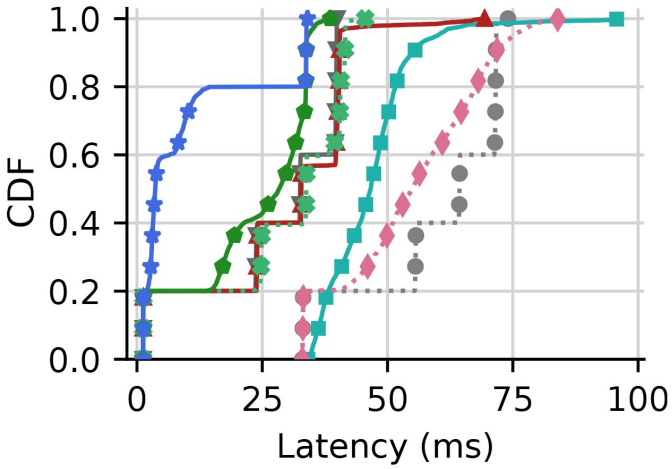
- (emulated global GCP results also available)

Throughput & Read/Write Latency

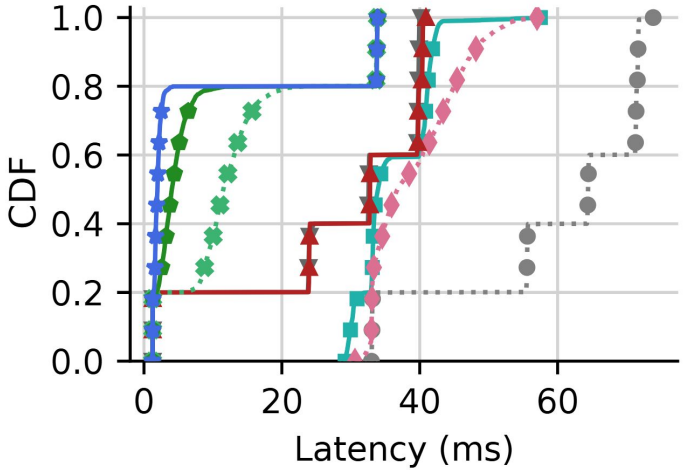


1% writes, 128B payloads; results grouped by client location affinity

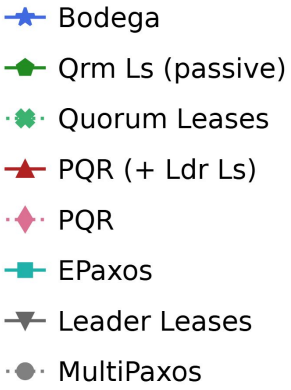
Read Latency CDFs



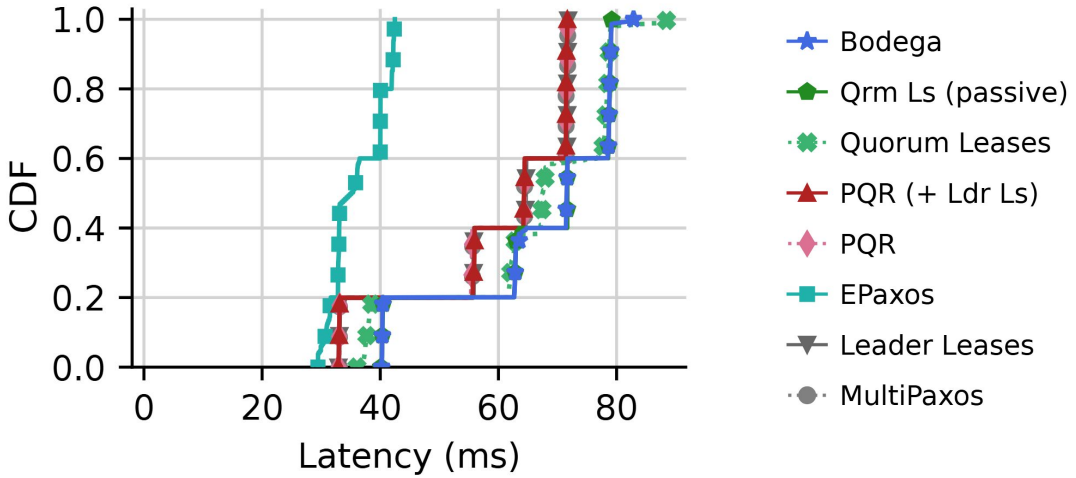
10% writes



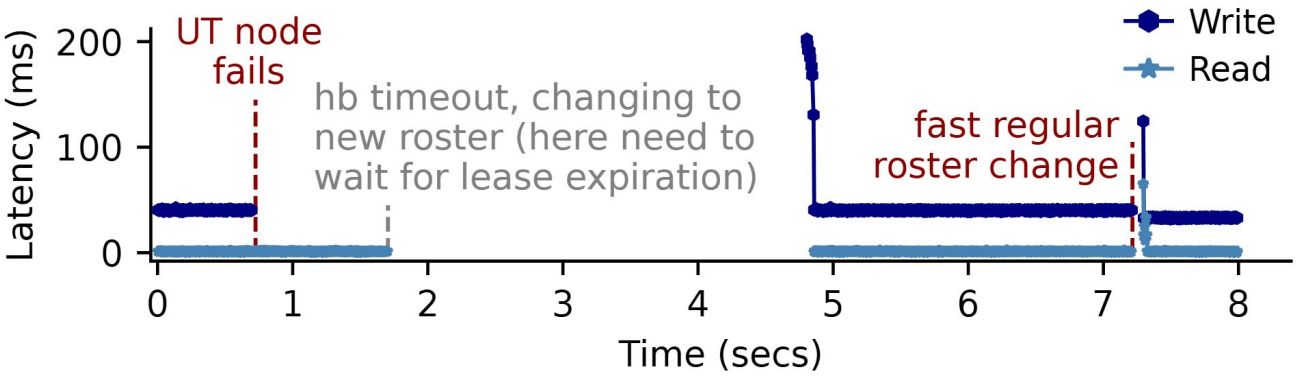
1% writes



Write Latency CDF

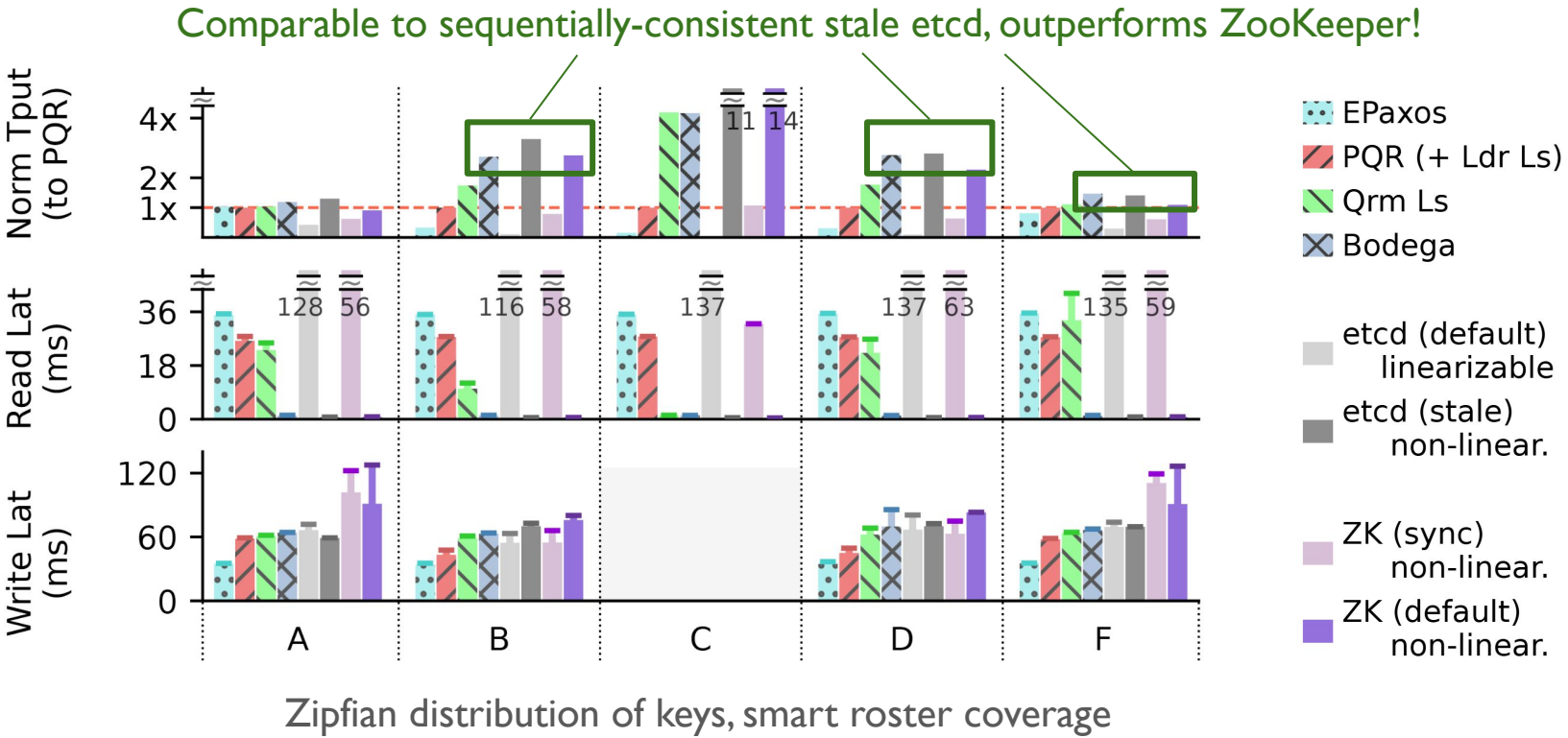


Roster Change Duration



lease timeout set to ~2.6 secs

YCSB Macro-benchmark against etcd, ZooKeeper



Zipfian distribution of keys, smart roster coverage

Outline

Problem Context	Main Contributions	System Implementation	Supportive Contributions	Discussions & Future Work
Consensus & SMR	*Optimistic Connectivity*	KV Testbed: **Summerset**	TLA+ Specifications	Related Work Areas
Classic Protocols	Protocol: **Crossword**	Linearizability Checking	Consistency Models Unified	Future Work
Cloud-Era Challenges	Protocol: **Bodega**		Jepsen Checker	

Summerset Key-Value Store

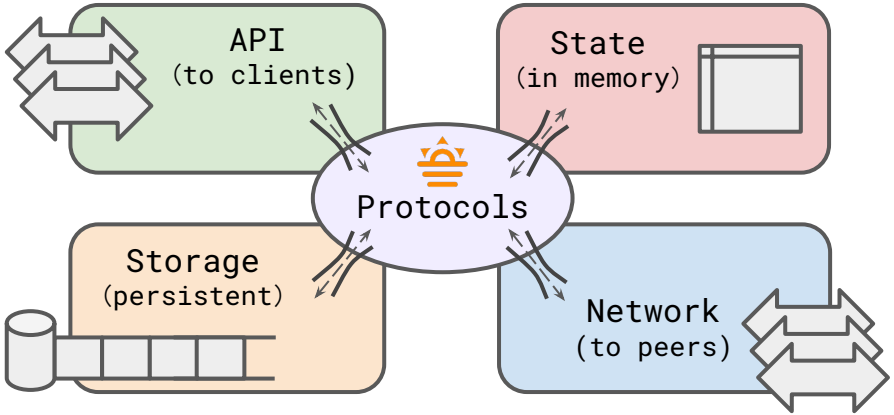
Summerset Protocol-Generic Key-Value Store

Available at: https://github.com/josehu07/summerset

Goals:

- Simple yet expressive replication system framework for implementing and evaluating consensus protocols; surprisingly, no such ready-to-use codebase found
- Concise coding of consensus protocols that reflect algorithm logic
- Performance and safety utilizing modern concurrent programming techniques

Summerset Server Node Architecture



“ Synchronization by
(low-cost) communication
(of ownership transfers). ”

Async Rust: `tokio` runtime
- user-level green threads (tasks)
- memory safety
- concurrency safety
- testing tooling

Modularized design
- channel-based synchronization
- zero explicit usage of `Mutex`
- each protocol
  == an event-loop module

Current Codebase Status

Protocol	LoC
RepNothing	0.5k
SimplePush	0.7k
ChainReplication	1.0k
MultiPaxos	3.0k
EPaxos	2.9k
Raft	2.0k
RSPaxos	2.3k
CRaft	2.3k
Crossword	3.4k
QuorumLeases	3.1k
Bodega	3.1k
Infrastructure	14.6k

Linearizability Checking

Linearizability Checker Implementation

Demo available at: https://github.com/josehu07/linearize

Effective and understandable *online* linearizability checker in Rust

- see Porcupine [6] for Golang
- assumes a known number of clients

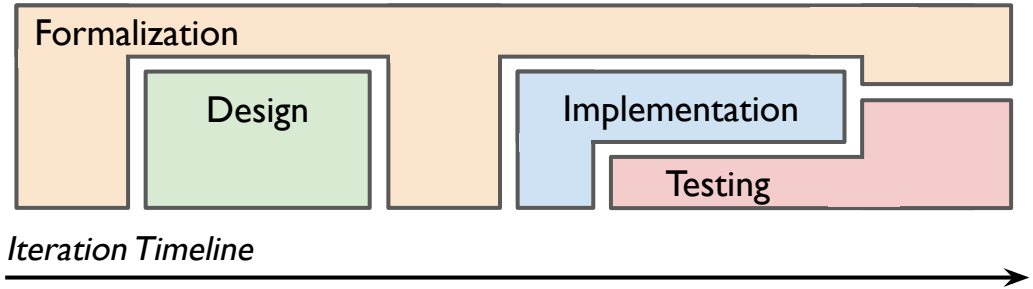
Outline

Problem Context	Main Contributions	System Implementation	Supportive Contributions	Discussions & Future Work
Consensus & SMR	*Optimistic Connectivity*	KV Testbed: **Summerset**	TLA+ Specifications	Related Work Areas
Classic Protocols	Protocol: **Crossword**	Linearizability Checking	Consistency Models Unified	Future Work
Cloud-Era Challenges	Protocol: **Bodega**		Jepsen Checker	

TLA+ Specifications: Formal & Practical

Why Formal Modeling?

How we incorporate formal methods in consensus research:



Benefits: helps more in the study & design phase than in model checking

- understand the problem context deeply (what do the assumptions really mean)
- forces a careful, thorough protocol design with well-defined steps

- model checking is the "tester" for the design against properties (mostly invariants)
- proof-based verification can further verify the implementation (future work)

Practical SMR-Style Specifications

Existing TLA+ models for consensus protocols tend to be single-decree and very abstract

=> loses the practical benefit of guiding design → implementation transition

Practical MultiPaxos Spec on https://github.com/tlaplus/Examples

MultiPaxos in SMR-Style	Guanzhou Hu			✔	✔

- Explicit log of commands
- Explicit client requests and messages
- Explicit linearizability and termination condition
- Explicit failure injection
- Asymmetric read/write quorums
- Leader leases and local read
- Commutative cluster of reads
- Crossword & Bodega specs built on top

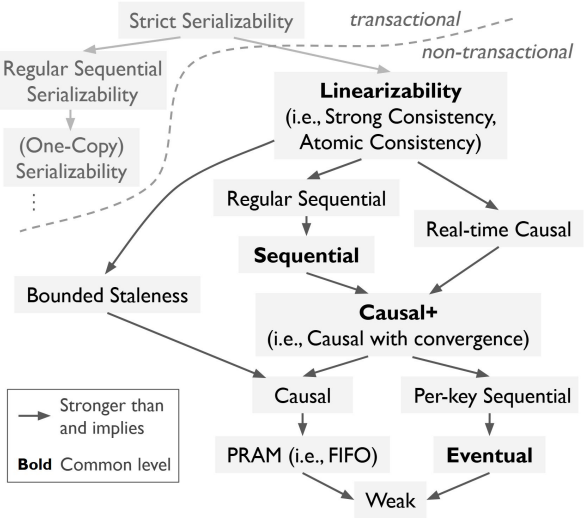
Unified Consistency Levels Hierarchy & Checker

Beyond Linearizability: Unified Consistency Hierarchy

No unified model of consistency levels except Viotti & Vukolić [7]

=> hard to comprehend weaker levels and their connection with linearizability

Shared Object Pool (SOP) model: common levels made unified and understandable



Consistency Level	Convergence	Relationship
Linearizability	SO	RT
Regular Sequential	SO	RT-W & CASL-R
Sequential	SO	CASL
Bounded Staleness	NPO	Bounded-CASL
Real-time Causal	CPO	RT′
Causal+	CPO	CASL
Causal	NPO	CASL
PRAM	NPO	FIFO
Per-key Sequential	CPO	CASL-per-key
Eventual	CPO	None
Weak	NPO	None

Multi-Level Consistency Checker

SOP model applied to consistency checking, yielding multi-level results

- Four common levels: linearizability, sequential, causal+, eventual
- Demonstrated with Jepsen framework integration [8]

System Setups		SOP-based Checker				Jepsen
System	Mode	Linr.	Seql.	Casl+	Evtl.	Knossos
etcd	Quorum read	●	●	●	●	Pass
	Stale read	○	●	●	●	No
	CAS as txns	○	●	●	●	No
ZK	Locked atoms	●*	●	●	●	Pass
	Local refs	○	○	●	●	No
RabbitMQ	P2P announce	○	○	○	●	No

Outline

Problem Context	Main Contributions	System Implementation	Supportive Contributions	Discussions & Future Work
Consensus & SMR	*Optimistic Connectivity*	KV Testbed: **Summerset**	TLA+ Specifications	Related Work Areas
Classic Protocols	Protocol: **Crossword**	Linearizability Checking	Consistency Models Unified	Future Work
Cloud-Era Challenges	Protocol: **Bodega**		Jepsen Checker	

Notable Related Work Areas

Notable Related Work Areas

Consensus Protocols & Replication Systems

- Erasure-coded consensus
- Keyspace partitioning
- Pipelining / chain structures
- Data dissemination
- Leaderless / multi-leader
- Distributed leases
- Atypical quorum assembly
- Membership management
- Lazy ordering
- Fail-slow tolerance
- Programmability
- Hardware acceleration
- BFT & blockchains
- Relaxed consistency

Optimistic Design Techniques

- Optimistic concurrency control (OCC)
- Conflict resolution mechanisms & CRDT
- Speculative execution

Cloud Studies & Surveys

- Cloud workload studies
- Cloud architecture & technology surveys
- Trace collection & workload generation

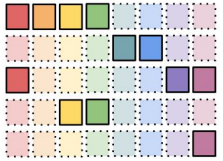
Testing & Formal Verification

- Controlled concurrency testing
- Formal modeling & specification tooling
- Formal verification with proofs

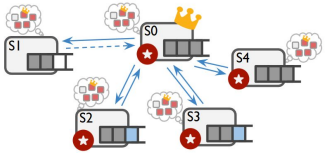
Future Work

Future Work

Deeper applications of the Optimistic Connectivity principle:



- Extending Crossword: asymmetric erasure-coded consensus



- Extending Bodega: general-purpose roster leases
- Optimistic quorums enabled by modern hardware semantics
  - Hardware-synchronized clocks
  - RDMA
  - Smart switches
  - Disaggregated, cache-coherent shared memory (CXL)
  - …

Future Work

Consensus solutions + advancements in other fields:

- Smart run-time policy making powered by ML
- Formally-proved implementation of modern replication systems
- Formally-verifiable analysis of performance metrics
- Visualization and observability

Conclusion

We present two cloud consensus protocols

Crossword – adaptive erasure-coded consensus for dynamic data-heavy workloads

Bodega – local linearizable reads in geo-scale consensus via lease-protected roster

that follow the principle of *Optimistic Connectivity*

Conclusion

We present two cloud consensus protocols

Crossword

Bodega

that follow the principle of *Optimistic Connectivity*

to solve

modern cloud's "4D" challenges

- Density
- Distance
- Diversity
- Dynamism

and develop a dependable infrastructure for consensus research

Summerset
KV testbed

Unified consistency levels hierarchy

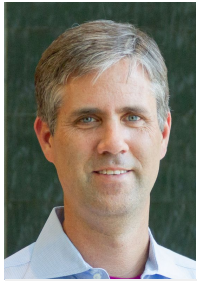
Practical TLA+ Specifications

Acks


Jiacheng Yu



Andrea Arpaci-Dusseau
Remzi

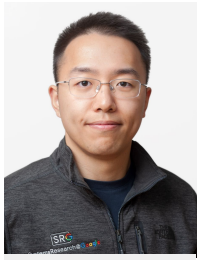

Mike Swift


Xiangyao Yu

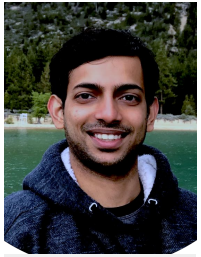

Tej Chajed


Kassem Fawaz


Yiwei Chen


Kan Wu


Jing Liu


Anthony Rebello


Kaiwei Tu


Yifan Dai


Vinay Banakar


Tingjia Cao


Xiangpeng Hao


Chenhao Ye

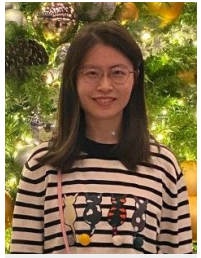

Shawn Zhong

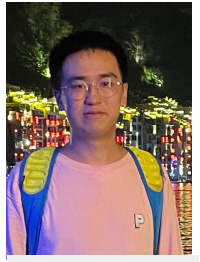

Suyan Qu


Sambhav Satija


John Shawger


Abigail Matthews


Wenjie Hu


Junxuan Liao


Jinlang Wang

Thank you!

Me:

Guanzhou Hu

josehu07

josehu.com

guanzhou.hu

References

1. Icons in slides: https://www.flaticon.com
2. Zhaoguo Wang et al. On the parallels between paxos and raft, and how to port optimizations. PODC ’19
3. Sanket Chintapalli et al. Pacemaker: When zookeeper arteries get clogged in storm clusters. CLOUD ’16
4. Sarah Tollman et al. EPaxos Revisited. NSDI ’21
5. Jovan Stojkovic et al. SmartOClock: Workload- and Risk-Aware Overclocking in the Cloud. ISCA ’24
6. Porcupine checker: https://github.com/anishathalye/porcupine
7. Paolo Viotti and Marko Vukolić. Consistency in Non-Transactional Distributed Storage Systems. CSUR ’16
8. Jepsen framework: https://github.com/jepsen-io/jepsen
9. Shuai Mu et al. When paxos meets erasure code: reduce network and storage cost in state machine replication. HPDC ’14
10. Zizhong Wang et al. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage and Network Cost. FAST ’20
11. Iulian Moraru et al. There Is More Consensus in Egalitarian Parliaments. SOSP ’13
12. Aleksey Charapko et al. Linearizable Quorum Reads in Paxos. HotStorage ’19
13. Maurice Herlihy. Dynamic quorum adjustment for partitioned data. TODS ’87
14. Heidi Howard et al. Flexible Paxos: Quorum intersection revisited. arXiv ’16
15. Leslie Lamport. Paxos made simple. 2001
16. Brian Oki and Barbara Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. PODC ’88
17. Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. ATC ’14

Backup Slides

Crossword

Real-time Estimate of Size-Time Mappings

Leader maintains a real-time updated simple linear regression model per follower s

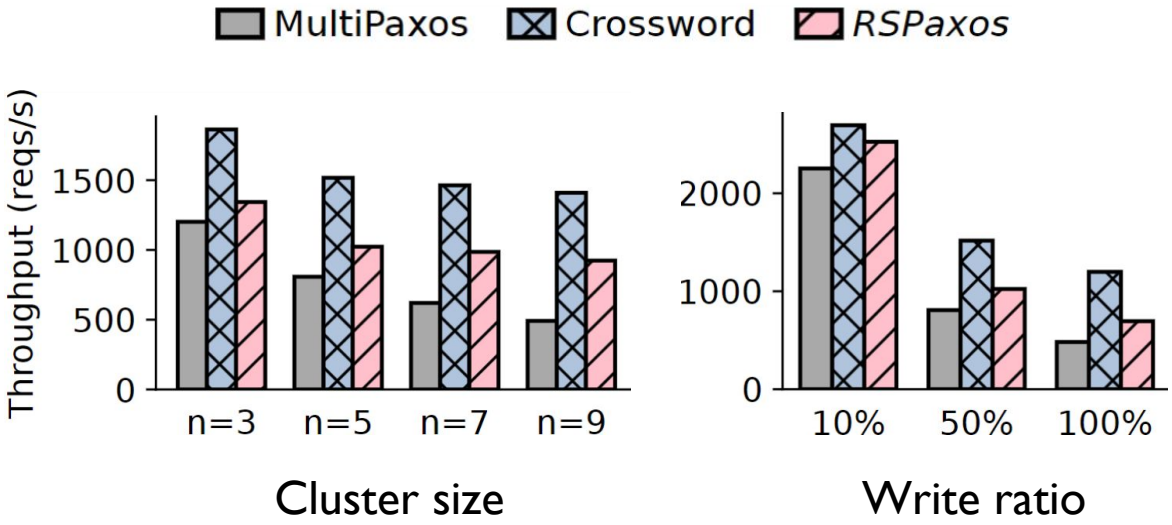
$$t_s(v) = d_s + 1/b_s * v$$

- Collects sizes of Accept and Heartbeat messages, and their round-trip times
- Maintains an online-updated model using datapoints collected in last 5 seconds
  - d_s ≈ real-time network RTT between leader and follower s
  - b_s ≈ real-time network bandwidth (rate) between leader and follower s

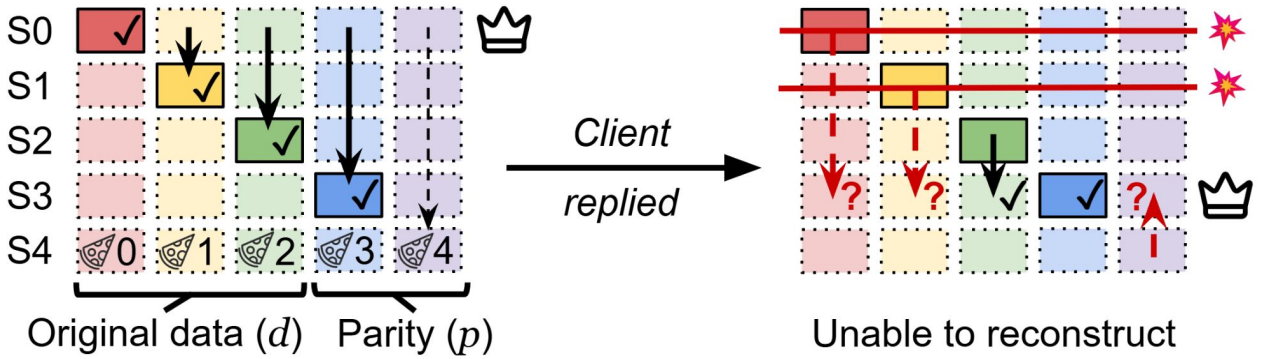
For an incoming request batch of size p, leader chooses the best c that:

minimizes the $(n\text{-}c)$-th smallest value of array $\{t_1(p/c), ..., t_{n-1}(p/c)\}$

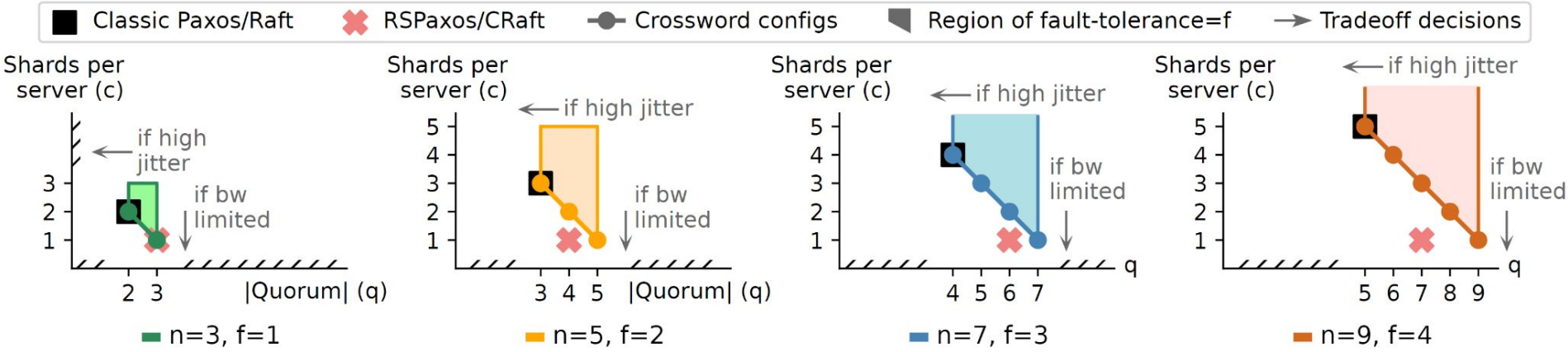
Varying Cluster Size & Write Ratio



Concurrent Failures Figure



All Constraint Boundary Figures



Constraint in the General Form

Denote the set of AcceptReplies received from followers as an *acceptance pattern*, ap

- $Nodes(ap)$ denotes the number of replies in ap, i.e., how many server nodes have replied (including the leader)
- $Cover(ap)$ denote the shard coverage of ap, which is the number of distinct shards that the replies cover
- $SubCover(ap, f)$ denote the subset coverage of ap, i.e., the minimum coverage among subsets of ap with f replies removed, where f is #failures tolerated

General constraint (still easily checkable programmatically):

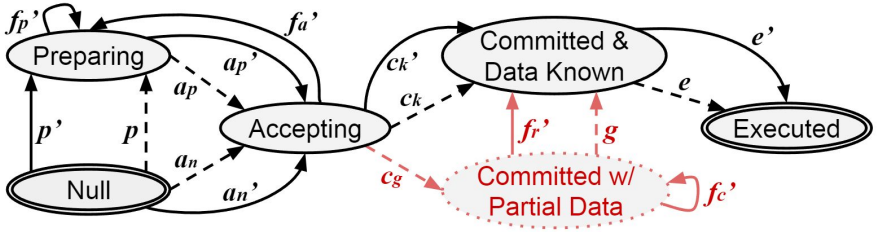
$$Nodes(ap) \geq m \quad \wedge \quad SubCover(ap, f) \geq d$$

Protocol Status Diagram

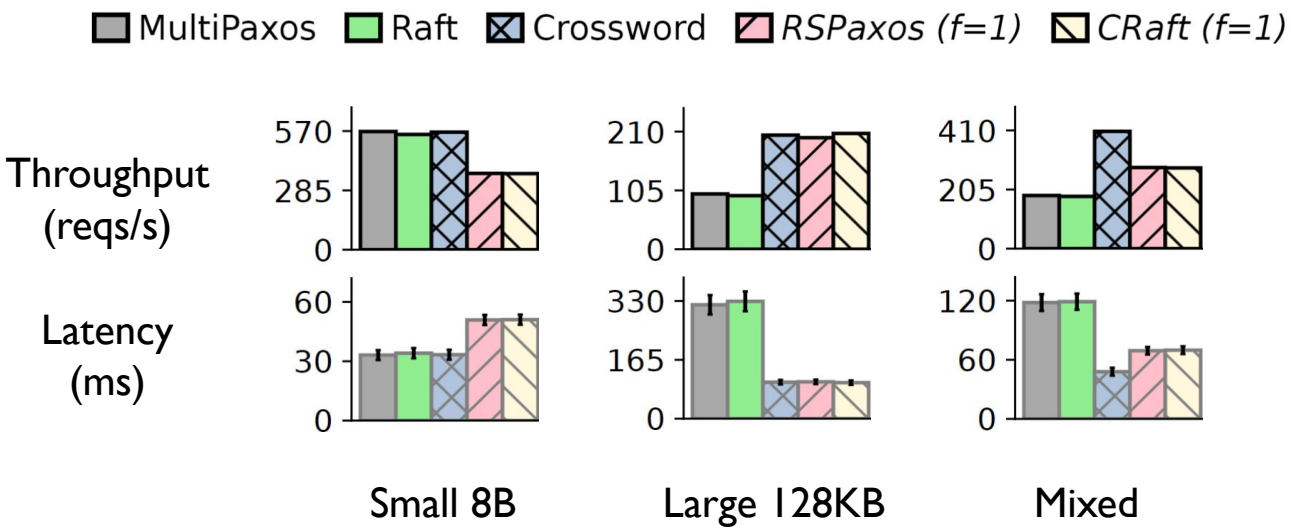
→	Trigger on leader	Action by leader
p'	client req, unprepared	broadcast `Prepare`
a_p'	decide prepared value	broadcast `Accept`, each w/ (a subset of) shards
a_n'	client req, ballot already prepared	broadcast `Accept`, each w/ (a subset of) shards
c_k'	reach commit condition	commit instance
e'	instance committed	execute, reply to client
f_p'	new leader after failover	redo with higher ballot
f_a'	new leader after failover	redo with higher ballot

- →	Trigger on followers	Action by followers
p	recv `Prepare`	send `Prepare` reply
a_p	recv `Accept`	send `Accept` reply
a_n	recv `Accept`	send `Accept` reply
c_k	leader committed, payload fully known	commit instance
e	committed, full payload	execute commands

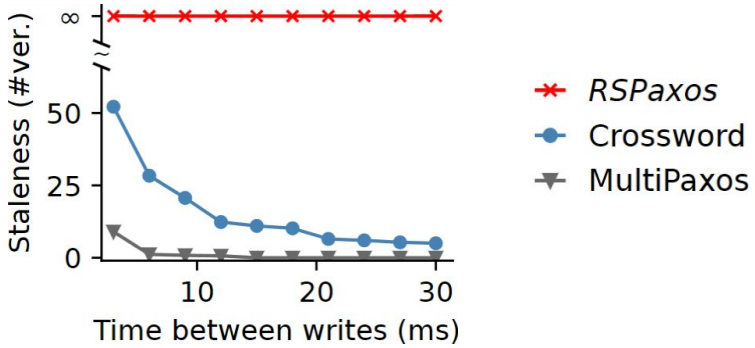
⇢ / - →	New gossiping-related transitions	
f_c'	new leader after failover	do reconstruction reads
f_r'	enough shards received	re-assemble the payload
c_g	leader committed, payload partially known	commit instance, schedule gossiping
g	enough shards gossiped	re-assemble the payload



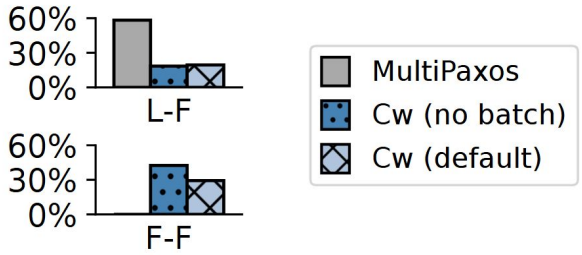
Critical Path Performance (WAN)



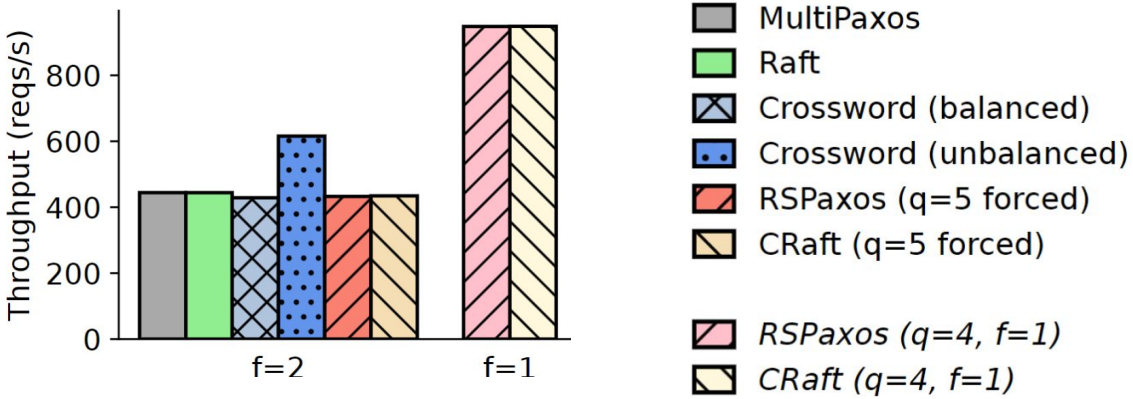
Follower Read Version Staleness



Bandwidth Usage w/ or w/o Gossip Batching



Unbalanced Case Performance



Durable Log Storage Space

Protocol	Storage Space Consumption
MultiPaxos Raft	~ 1467MB
Crossword RSPaxos CRaft	~ 913MB

YCSB-A Macro-benchmark

Keyspace partitioning

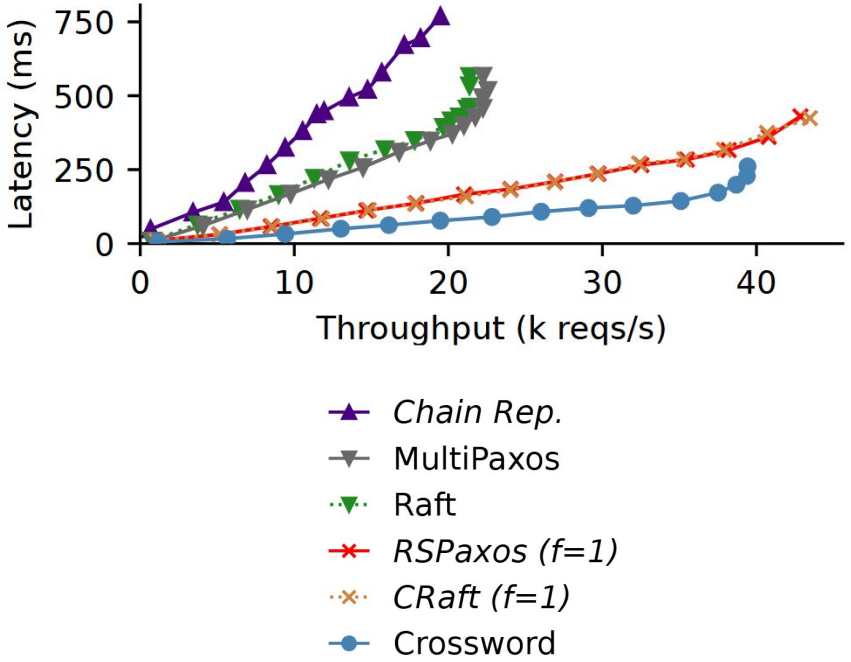
- split keys into 5 partitions, each a separate consensus cluster
- machine i runs the leader of partition i and the followers of the rest

YCSB-A trace

- 50% reads + 50% writes

TiDB payload profile

- for writes, payload sizes drawn from the TiDB payload profile



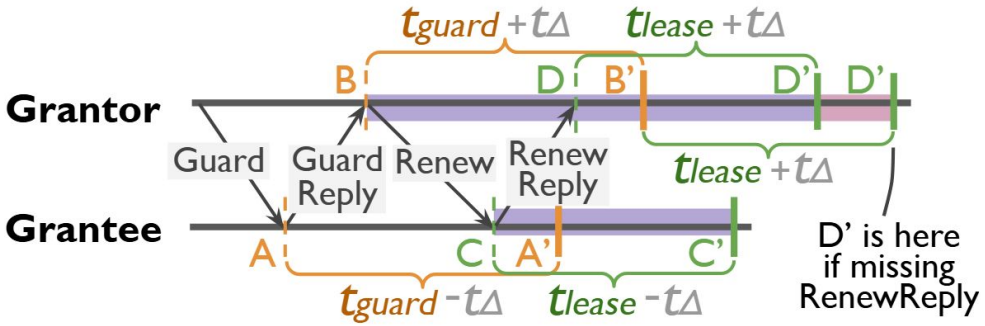
RS Code Computation Overhead

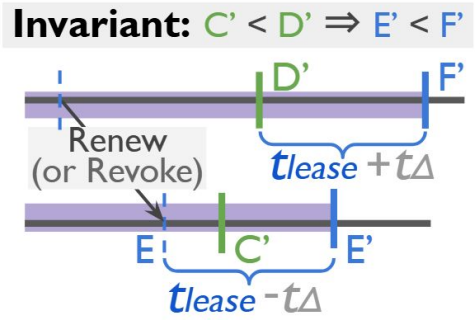
Payload size	4KB	16KB	64KB	256KB	1MB	4MB
Time taken	1µs	4µs	16µs	77µs	1ms	6ms
CPU usage	1.25%	1.24%	1.24%	1.26%	1.25%	1.26%
Memory usage	1.6KB	6.4KB	25.6KB	102.4KB	409.6KB	1.6MB

RS $(5,3)$-coding computation time and resource usage overhead.

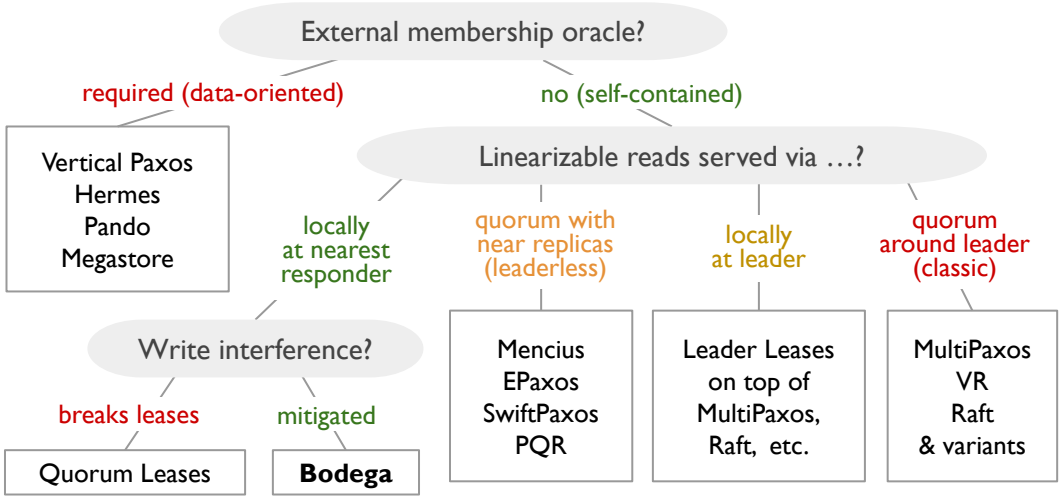
Bodega

Distributed Leases Explained

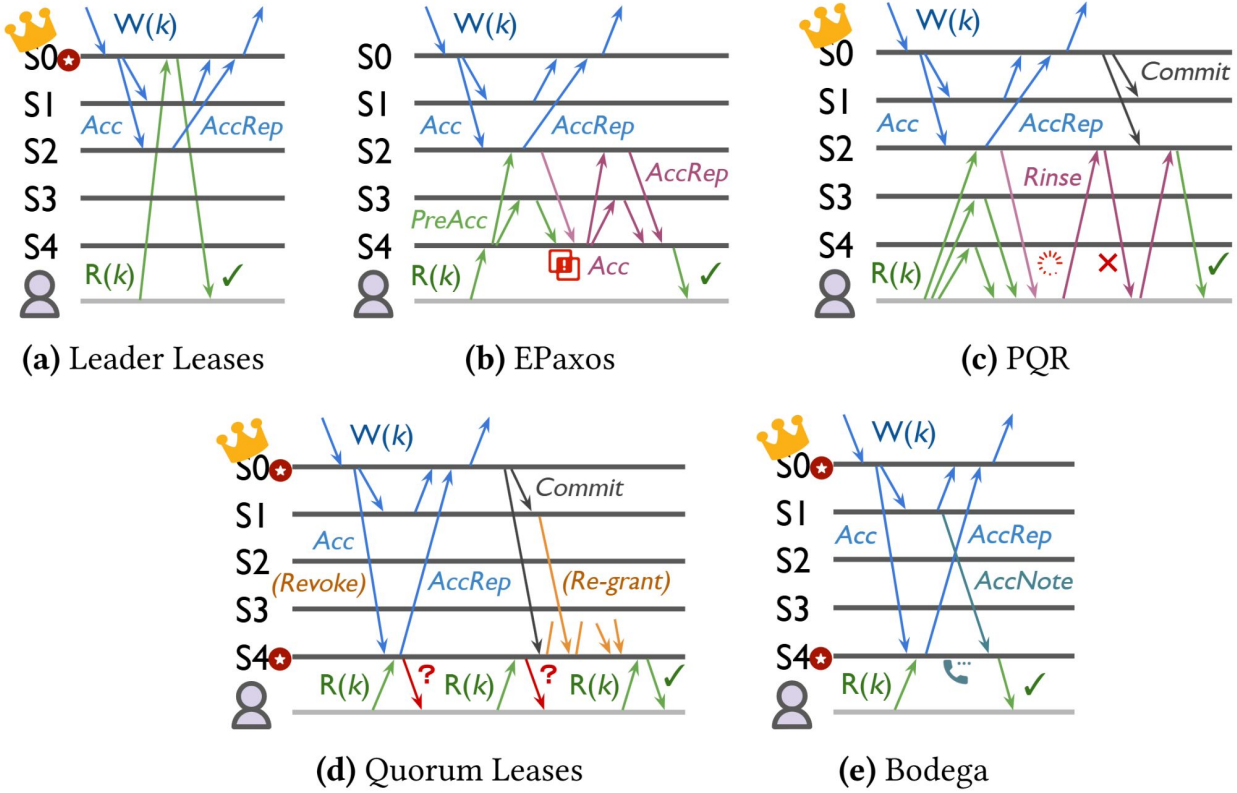




Related Work Categorization



Related Work Timeline Figures



(a) Leader Leases (b) EPaxos (c) PQR

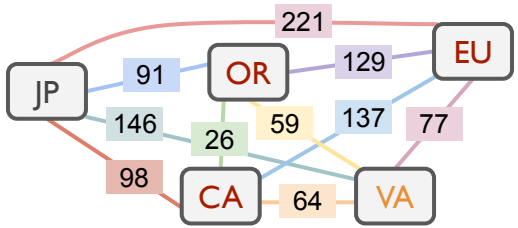
(d) Quorum Leases (e) Bodega

Qualitative Comparison

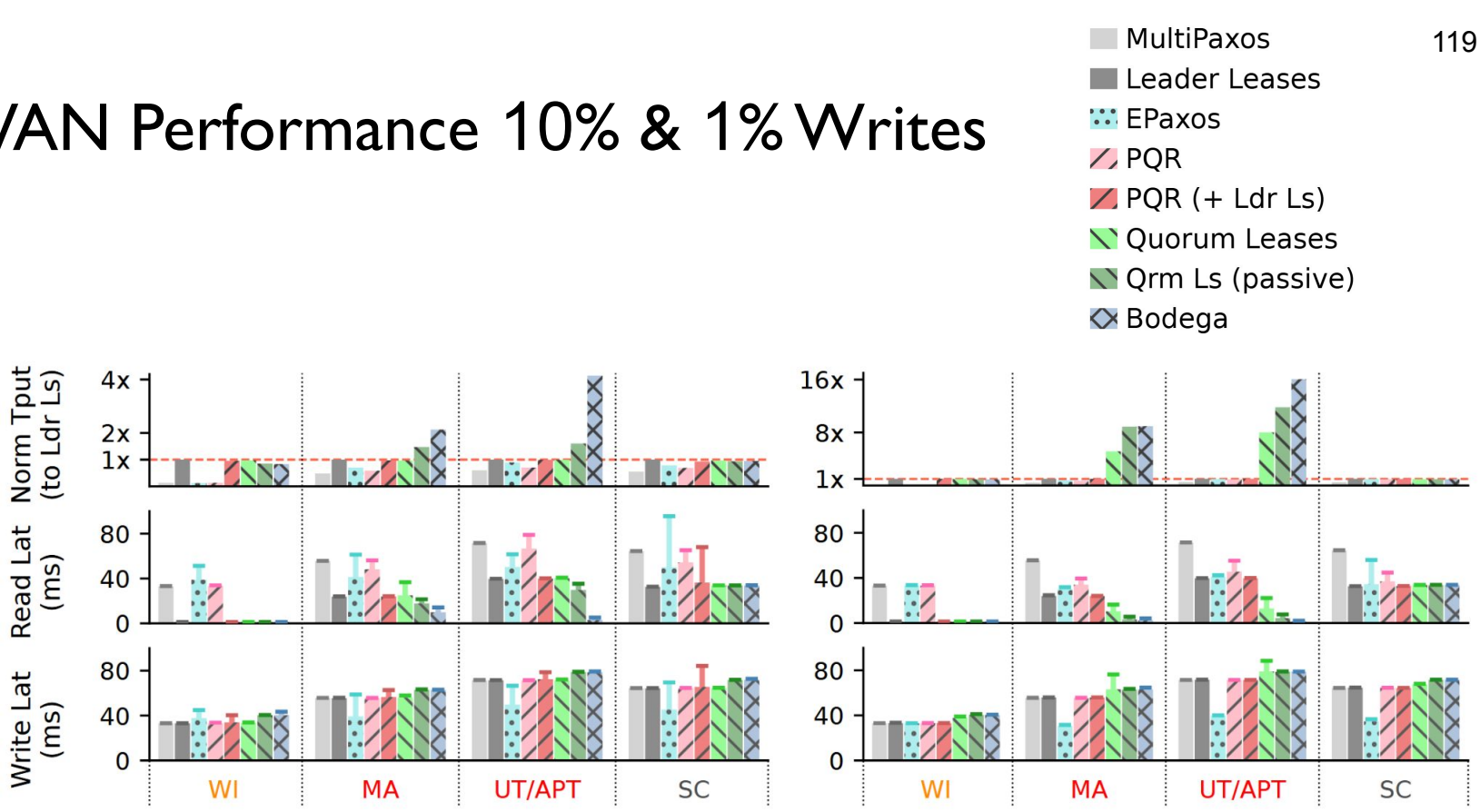
Protocol	W	R	R*	D*	⚡	⬚
Leader Ls	$l+m$	l	l	-	●	○
EPaxos	$c+M$	$c+M$	$c+M+m$	M	●	○
Hermes	$c+N$	c	$c \sim c+\frac{N}{2}$	$\frac{N}{2}$	◐	○
PQR	$l+m$	$c+m$	$c+m*\text{rinses}$	m	●	○
PQR (+ Ldr Ls)	$l+m$	$c+m$	$c+m+l$	m	●	○
Pando	$c+m$	$c+m$	$c+N$	N	◐	◐
Megastore	$l+2N$	c	$c+l$	$2N$	◐	○
Quorum Ls	$l+N$	c	$c+l$	$2N$	●	●
Qrm Ls (passive)	$l+N$	c	$c+l$	N	◐	◐
Bodega	$l+N$	c	$c \sim c+\frac{m}{2}$	$\frac{m}{2}$	●	●

Table 4.1: Qualitative comparison across protocols *assuming the most read-optimized roster configuration of each protocol. Metrics are* **W***: write latency;* **R***: read latency if quiescent;* **R***: read latency if there is an interfering write;* **D***: read performance degradation period length.* ⚡*: fault tolerance (without external oracle).* ⬚*: allows tunable rosters. See §4.3.1 for the explanation of metric values. Cells are shaded darker if their example numeric values are higher using Fig.4.8(b) GEO setting numbers.*

GEO Setting RTTs



WAN Performance 10% & 1% Writes



(e) WAN, 10% writes

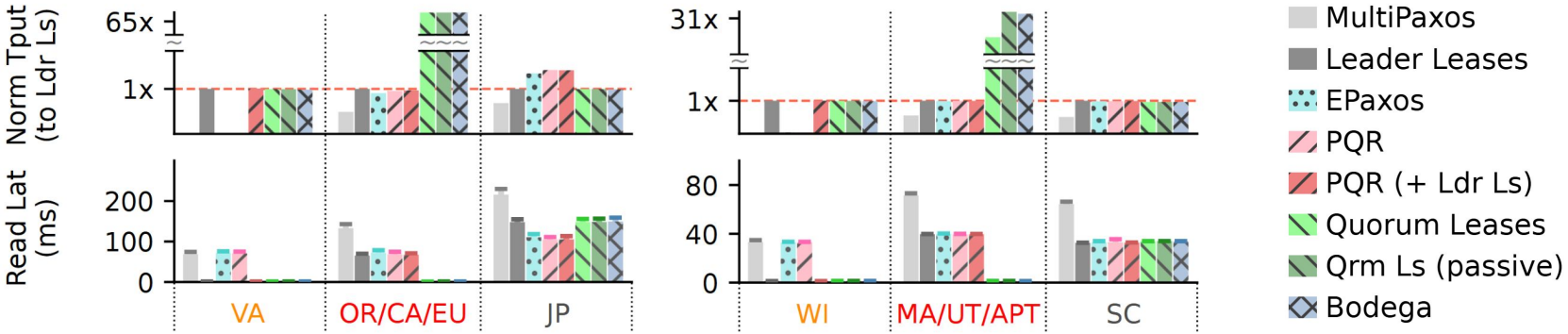
(f) WAN, 1% writes

GEO Performance 10% & 1% Writes

(a) GEO, 10% writes

(b) GEO, 1% writes

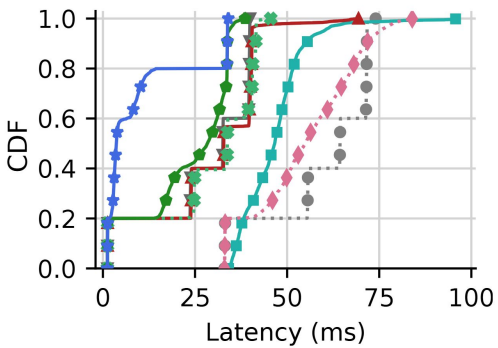
Read-Only Performance



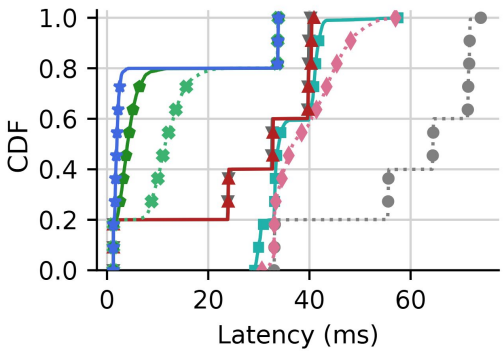
(c) GEO, 0% write (read-only)

(d) WAN, 0% write (read-only)

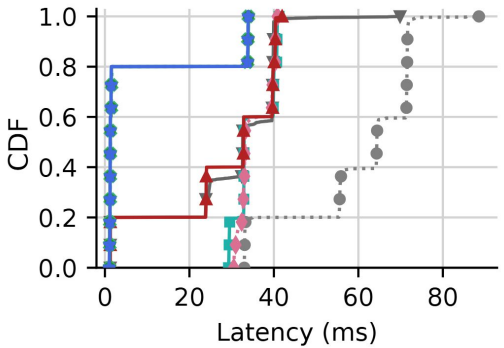
Read Latency CDFs



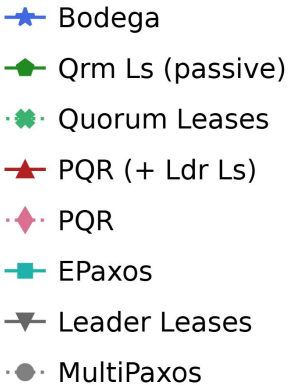
10% writes



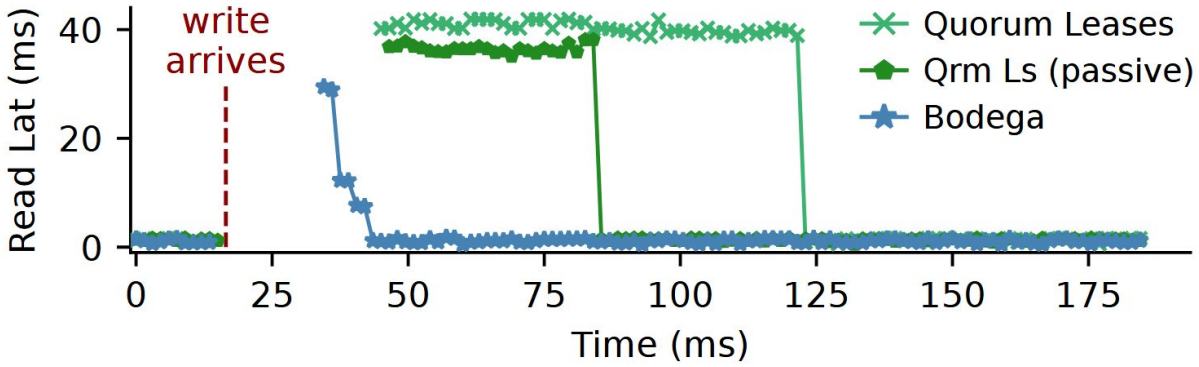
1% writes



0% writes



Visualizing a Write Interference



Throughput vs. Write Ratio or Payload Size

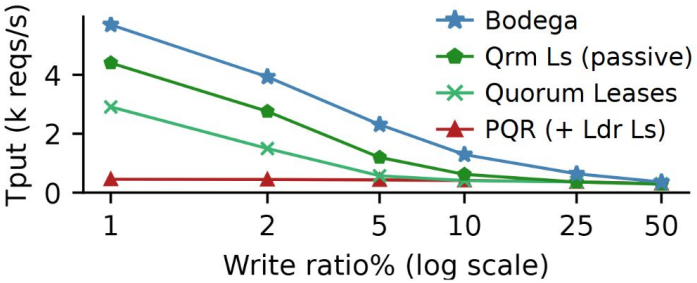


Figure 4.12: Throughput vs. write ratio. *x-axis is log-scale (same for Fig. 4.13). See §4.5.2.*

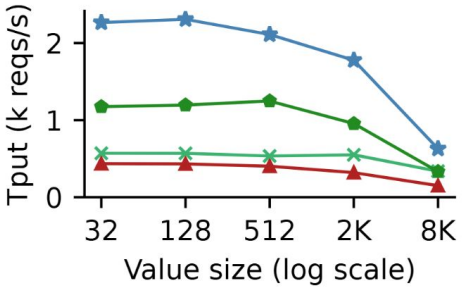
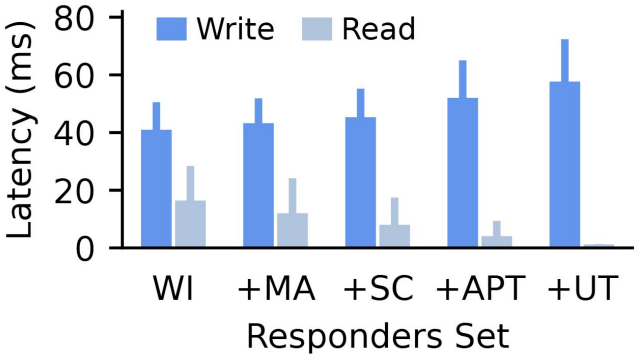
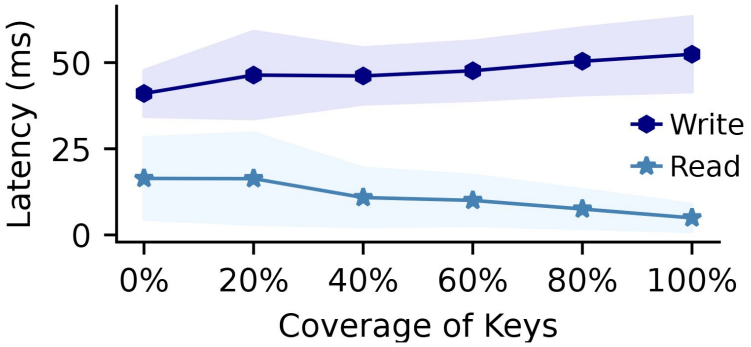


Figure 4.13: Throughput vs. payload size. *See §4.5.2.*

Roster Coverage

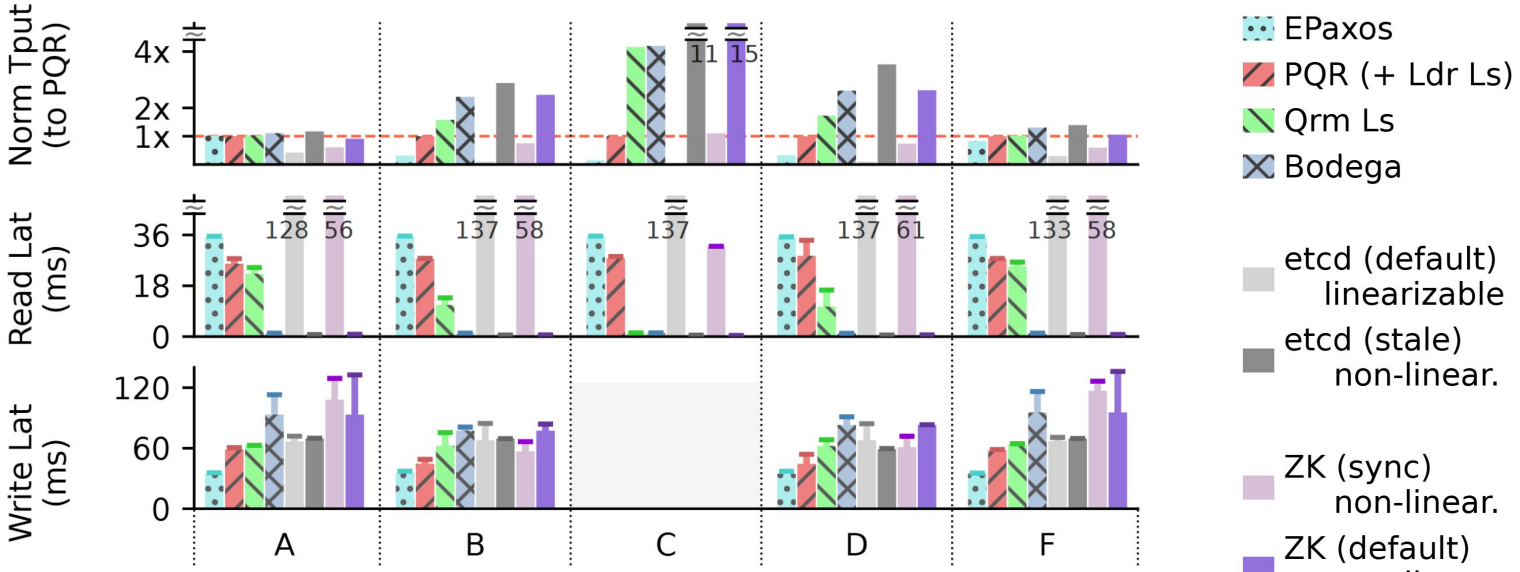


Increasing selection of responders for a key



Increasing range of keys that have responder

YCSB with Uniform Distribution



(a) Uniform, full-coverage roster

Questions & Answers

Limitations of this research? Weakness?

- Crossword: not enough adaptability evaluation
- Bodega: not enough failure resilience evaluation
- Evaluation environment all on CloudLab – may not reflect the true production cloud
- Implementation not formally verified
- Integration with well-renowned systems could be pushed further
- Usage of modern hardware (sync. clock, RDMA, smart switch, CXL) not explored

Crossword

Why use Balanced RR assignments like this?

- Maximizes shard span across any given subset of replies

Crossword versus two-layer metadata-data separation architecture:

- Crossword is a flat deployment of consensus, simpler and easier to do (comments from Cockroach folks), reduces system complexity
- Metadata workloads themselves can become heavy – infinite layering
- Requires two rounds of communication, one to the data store and one to the metadata store, in order => longer latency?
- Requires the data store to provide multi-versioning semantics

Bodega

Why optimistic holding still counts as local read?

- Because no active action is taken by the responder. It serves the reads as soom as the commit notification arrives, which could be within nanoseconds.

External coordination service drawbacks:

- Need to reach the coordination service for roster queries, slow
  - Unless leases are deployed between the coordination service and the replicas, but not sure how
- Not self-contained, extra complexity
- Extra resource cost
- Reduces fault-tolerance to that of the coordination service

Miscs

Partial synchrony vs. asynchrony?

- asynchronous network + randomized timeouts == fine
- otherwise, liveness is impossible to guarantee, unless partially synchronous

What about dynamic quorums [13]:

- Static pre-determined quorum members, not run-time tunable