Dissecting the Performance of Strongly-Consistent Replication Protocols

Ailidani Ailijiang[*]
Microsoft
Ailidani.Ailijiang@microsoft.com

Aleksey Charapko[†]
University at Buffalo, SUNY
acharapk@buffalo.edu

Murat Demirbas[†]
University at Buffalo, SUNY
demirbas@buffalo.edu

ABSTRACT

Many distributed databases employ consensus protocols to ensure that data is replicated in a strongly-consistent manner on multiple machines despite failures and concurrency. Unfortunately, these protocols show widely varying performance under different network, workload, and deployment conditions, and no previous study offers a comprehensive dissection and comparison of their performance. To fill this gap, we study single-leader, multi-leader, hierarchical multi-leader, and leaderless (opportunistic leader) consensus protocols, and present a comprehensive evaluation of their performance in local area networks (LANs) and wide area networks (WANs). We take a two-pronged systematic approach. We present an analytic modeling of the protocols using queuing theory and show simulations under varying controlled parameters. To cross-validate the analytic model, we also present empirical results from our prototyping and evaluation framework, Paxi. We distill our findings to simple throughput and latency formulas over the most significant parameters. These formulas enable the developers to decide which category of protocols would be most suitable under given deployment conditions.

ACM Reference Format:
Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3299869.3319893

[*]Work completed at University at Buffalo, SUNY.
[†]Also with Microsoft, Redmond, WA.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319893

1 INTRODUCTION

Coordination services and protocols play a key role in modern distributed systems and databases. Many distributed databases and datastores [4, 7–10, 12, 13, 16, 18, 23, 24, 31, 40] use consensus to ensure that data is replicated in a strongly-consistent manner on multiple machines despite failures and concurrency.

Fault-tolerant distributed consensus problem is addressed by the Paxos [25] protocol and its numerous variations and extensions [1, 19–21, 26, 30, 33–35]. The performance of these protocols become important for the overall performance of the distributed databases. These protocols show widely varying performance under different conditions: network (latency and bandwidth), workload (command interference and locality), deployment size and topology (LAN/WAN, quorum sizes), and failures (leader and replica crash and recovery). Unfortunately, there has been no study that offers a comprehensive comparison across consensus protocols, and that dissects and explains their performance.

1.1 Contributions

We present a comprehensive evaluation of consensus protocols in local area networks (LANs) and wide area networks (WANs) and investigate many single-leader, multi-leader, hierarchical multi-leader and leaderless (opportunistic leader) consensus protocols. We take a two-pronged systematic approach and study the performance of these protocols both analytically and empirically.

For the analytic part, we devise a queuing theory based model to study the protocols controlling for workload and deployment characteristics and present high-fidelity simulations of the protocols. Our model captures parameters impacting throughput, such as inter-node latencies, node processing speed, network bandwidth, and workload characteristics. We made the Python implementations of our analytical models available as opensource.

For our empirical study, we developed Paxi, a prototyping and evaluation framework for consensus and replication protocols. Paxi provides a leveled playground for protocol evaluation and comparison. The protocols are implemented using common building blocks for networking, message handling, quorums, etc., and the developer only needs to fill in two modules for describing the distributed coordination protocol. Paxi includes benchmarking support to evaluate the protocols in terms of their performance, availability, scalability, and consistency. We implemented Paxi in Go [17] programming language and made it available as opensource at https://github.com/ailidani/paxi.

The analytical model and the Paxi experimental framework are complementary. The Paxi experiments cross-validate the analytical model. And the analytical model allows exploring varying deployment conditions that are difficult to arrange and control using the experimental framework.

1.2 Results

Armed with both the simulation results from the analytical model and the experimental results obtained from the Paxi platform implementations, we distill the performance results into simple throughput and latency formulas and present these in Section 6. These formulas provide a simple unified theory of strongly-consistent replication in terms of throughput —Formula 3: $L/(1 + c)(Q + L - 2)$— and latency —Formula 7:$(1+c)*((1-l)*(D_L+D_Q)+l*D_Q)$. As such, these formulas enable developers to perform back-of-the-envelope performance forecasting. In Section 6 we discuss these results in detail and provide a flowchart to serve as a guideline to identify which consensus protocols would be suitable for a given deployment environment. Here we highlight some significant corollaries from these formulas.

Protocol parameters	L	number of leaders
	Q	quorum size
Workload parameters	c	conflict probability
	l	locality
Deployment parameters	D_L	latency to leader
	D_Q	latency to quorum

Considering protocol parameters, an effective protocol-level revision for improving throughput and latency is to increase the number of leaders in the protocol, while trying to avoid an increase on the number of conflicts. Increasing the number of leaders is also good for availability: In Paxos, failure of the single leader leads to unavailability until a new leader is elected, but in multi-leader protocols most requests do not experience any disruption in availability, as the failed leader is not in their critical path. Another protocol revision that helps to improve throughput and latency is to reduce Q, the quorum size, provided that fault-tolerance requirements are still met.

As workload parameters are concerned, reducing conflict probability and increasing locality (in the presence of multiple leaders) are beneficial. However, there is an interplay between the number of leaders and probability of conflicts: increasing the number of leaders (which helps for throughput and latency) may cause an increase on conflicts (which hurts throughput and latency). EPaxos [30] protocol suffers from this problem. Multi-leader protocols that learn and adapt to locality, such as WPaxos [1] and WanKeeper [2], are less susceptible to this problem.

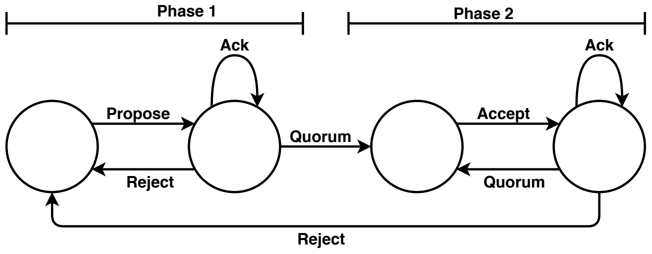


Figure 1: State transitions for two-phase coordination

Finally, the deployment parameters, distance to the leader and distance from leader to the quorum number of nodes, also have a big effect on the latency in WAN deployments. Note that these deployment parameters shadow the protocol parameters, the number or leaders and the quorum size. In WANs, other factors also affect latency. The asymmetric distances between datacenters, the access pattern locality, and unbalanced quorum distances complicate forecasting the performance WAN deployments.

1.3 Outline of the rest of the paper

In Section 2 we briefly introduce the protocols we study. We discuss our analytical model in Section 3 and our prototyping/evaluation framework in Section 4. We present the evaluation in Section 5, discussion of the findings in Section 6, and conclude the paper in Section 7.

2 PROTOCOLS

Many coordination and replication protocols share a similar state transition pattern as shown in Figure 1. These protocols typically operate in two phases. In the *phase-1*, some node establishes itself as a leader by announcing itself to other nodes and gaining common consent. During this stage, an incumbent leader also acquires information related to any prior unfinished commands in order to recover them in the next phase. The *phase-2* is responsible for replicating the state/commands from the leader to the nodes.

Leveraging this two phase pattern, we give brief descriptions of the protocols in our study below. These protocols provide strong consistency guarantees for data replication in distributed databases.

Paxos. The Paxos consensus protocol [25] is typically employed for realizing a fault-tolerant replicated state machine (RSM), where each node executes the same commands in the

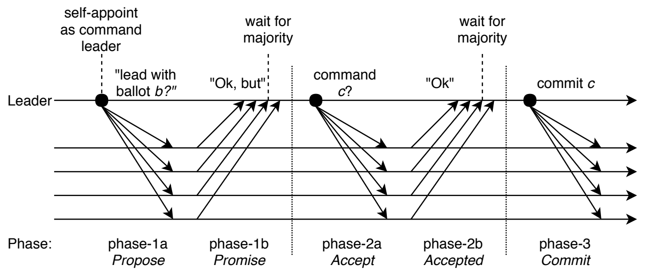


Figure 2: Overview of Paxos algorithm

same order to arrive to identical states. Paxos achieves this in 3 distinct phases: propose (phase-1), accept (phase-2), and commit as shown in Figure 2. During phase-1, a node tries to become the leader for a command by proposing it with a *ballot number*. The other nodes acknowledge this node to lead the proposal only if they have not seen a higher ballot before. Upon reaching a majority quorum of acks in the propose phase, the "leader" advances to phase-2 and tells the followers to accept the command. The command is either a new one proposed by the leader, or an older one learned in phase-1. (If the leader learns some older pending/uncommitted commands in phase-1, it must instruct the followers to accept the pending commands with the highest ballot numbers.) Once the majority of followers acknowledge the acceptance of a command, the command becomes anchored and cannot be lost. Upon receiving the acks, the leader sends a commit message that allows the followers to commit and execute the command in their respected state machines.

Several optimizations are adopted over this basic scheme. The commit phase is typically piggybacked to the next message broadcasted from the leader, alleviating the need for an extra message. Another popular optimization, known as multi-Paxos or multi-decree Paxos [37], reduces the need for extra messages further by allowing the same leading node to instruct multiple commands to be accepted in different slots without re-running the propose phase, as long as its ballot number remains the highest the followers have seen. In the rest of the paper, we use Paxos to refer to the multi-Paxos implementation.

As examples of databases that uses Paxos, FaunaDB [16] uses Raft [33] (a Paxos implementation) to achieve consensus, Gaios [7] uses Paxos to implement a storage service, WANDisco [8] uses Paxos for active-active replication, Bizur [19] key-value store uses Paxos for reaching consensus independently on independent keys, pg_paxos [13] adopts Paxos for fault-tolerant, consistent table replication in PostgreSQL, and Clustrix [10] distributed SQL database uses Paxos for distributed transaction resolution.

Flexible Paxos. Flexible quorums Paxos, or FPaxos [20], observes that the majority quorum is not necessary in phase-1 and phase-2. Instead, FPaxos shows that the Paxos properties hold as long as all quorums in phase-1 intersect with all quorums in phase-2. This result enables deploying multi-decree Paxos protocols with a smaller quorum in phase-2, providing better performance at the cost of reduced fault tolerance.

Vertical Paxos. Vertical Paxos (VPaxos) [26] separates the control plane from the data plane. VPaxos imposes a master Paxos cluster above some Paxos groups in order to control any configuration changes, and enables a quick and safe transition between configurations without incurring any stop time: one Paxos group finishes the commands proposed in the previous configuration, while another Paxos group starts on the commands in the new configuration. The ability to safely switch configurations is useful in geo-replicated datastores, since it allows for relocating/assigning data/objects to a different leader node in order to adjust to changes in access locality. Spanner [12] and CockroachDB [24] are examples of databases that uses Paxos groups to work on partitions/shards with another solution on top for relocating/assigning data to another Paxos group.

WanKeeper. WanKeeper [2] is a hierarchical protocol composed of two consensus/Paxos layers. It employs a token broker architecture to control where the commands take place across a WAN deployment. The master resides at level-2 and controls all token movement, while the level-1 Paxos groups located in different datacenters across the globe, execute commands only if they have a token for the corresponding objects. When multiple level-1 Paxos groups require access to the same object, the master layer at level-2 retracts the token from the lower level and performs commands itself. Once the access locality settles to a single region, the master can pass the token back to that level-1 Paxos group to improve latency.

WPaxos. WPaxos [1] is a multi-leader Paxos variant designed for WANs. It takes advantage of flexible quorums idea to improve WAN performance, especially in the presence of access locality. In WPaxos, every node can own some objects and operate on these objects independently. Unlike Vertical Paxos, WPaxos does not consider changing the object ownership as a reconfiguration operation and does not require an external consensus group. Instead WPaxos performs object migration between leaders by carrying out a phase-1 across the WAN, and commands are committed via phase-2 within the region or neighboring regions. Since the process of moving ownership between leaders is performed using the core Paxos protocol, WPaxos operates safely without requiring an external master as in vertical Paxos or WanKeeper. FleetDB [9] adopts WPaxos for implementing distributed transactions over multiple datacenter deployments in WAN.

Egalitarian Paxos. Egalitarian Paxos [30], or EPaxos, is a leaderless solution, where every node can opportunistically become a leader for some command and commit it. When a command does not interfere with other concurrent commands, it commits in a single round after receiving the acks from a fast quorum, which is approximately 3/4ths of all nodes. In a sense, EPaxos compacts phase-2 to be part of phase-1, when there are no conflicts. However, if the fast quorum detects a conflict between the commands, EPaxos defaults back to the traditional Paxos mode proceeds with a second phase to establish order on the conflicting commands. The main advantage of EPaxos is the ability to commit non-interfering commands in just a single round trip time. This works well when the cluster operates on many objects, and the probability of any two nodes operating on the same objects is small. As an example of a leaderless approach like EPaxos, MDCC [23] has used Fast Paxos and Generalized Paxos (before EPaxos) to implement multi-datacenter strongly consistent replication.

3 PERFORMANCE MODEL

Modeling the performance of protocols provides an easy way to test different configurations or ideas. In this section, we develop our models for estimating latency and maximum throughput of the strong consistency replication protocols. Our models leverage queueing theory [3] and k-order statistics to account for various delays and overheads due to the message exchange and processing.

3.1 Assumptions

We make a number of assumptions about the machines and network in order to constrain the scope of our models and keep them simple and easy to understand.

We assume the round-trip communication latency (RTT) in the network between any two nodes to be normally distributed. This assumption simplifies the reasoning about the effects of network latency on the consensus performance. Exponential distribution is often used to model network latencies [6], however, our experiments conducted in AWS EC2 point to an approximately normal latency distribution in local area in AWS cluster, as shown in Figure 3.

We assume all nodes to have stable, uniform network bandwidth. Our models do not account for bandwidth variations among the nodes. For simplicity, we assume all nodes in the modeled cluster to be of identical performance. In particular, we assume identical CPU and network interface card (NIC) performance. We only consider the case of a single processing pipeline – our modeled machines have a single network card and CPU.

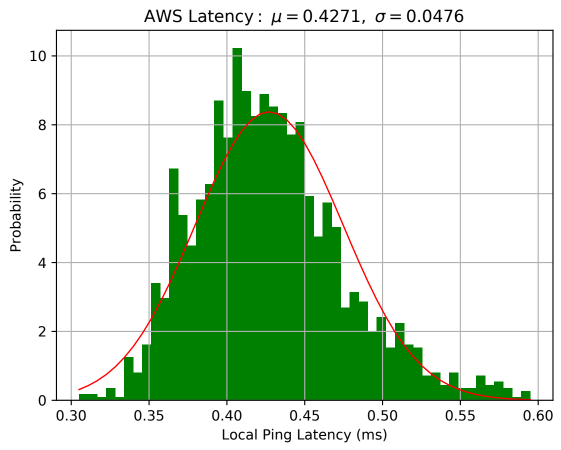


Figure 3: Histogram of local area RTTs within Amazon AWS EC2 region over a course of a few minutes

3.2 Simple Queueing Models

Queueing theory [3] serves as the basis for our analytical study of consensus protocols. We treat each node in a system as a single processing queue consisting of both NIC and CPU. The protocols operate by exchanging the messages that go through the queue and use the machine's resources. Sending a message out from a node requires some time to process at the CPU and some time to transmit through the NIC. Similarly, an incoming message first needs to clear the NIC before being deserialized and processed by a CPU. When a message enters a queue, it needs to wait for any prior messages to clear and resources to become available.

Queueing models enable estimating the average waiting time spent in the queue before the resources become available. To make such estimates, queueing models require just two parameters: service time and inter-arrival time. Service time describes how long it takes to process each item in the queue once it is ready to be consumed. Inter-arrival time controls the rate at which items enter the queue. With high inter-arrival time, new items come in rarely, keeping the queue rather empty. Low inter-arrival time causes the queue to fill up faster, increasing the chance of items having to wait for predecessors to exit the queue.

Since in our model messages immediately transition from CPU to NIC (or vice-versa) and have no possibility of leaving the system without bypassing either the CPU or NIC, we treat these two components as a single queue. This simplifies the "queueing network" significantly and facilitates our modeling.

For our purposes we have considered four different types of queue approximations: $M/M/1$, $M/D/1$, $M/G/1$ and $G/G/1$, where the first letter represents the inter-arrival assumption,

	Request Arrival	**Service Time**	W_q
M/M/1	Poisson Process rate λ	Exponential Distribution rate μ	$\frac{\rho^2}{\lambda(1-\rho)}$
M/D/1	Poisson Process	Constant s rate $\mu = 1/s$	$\frac{\rho}{2\mu(1-\rho)}$
M/G/1	Poisson Process	General Distribution	$\frac{\lambda^2\sigma^2+\rho^2}{2\lambda(1-\rho)}$
G/G/1	General Distribution	General Distribution	$\approx \frac{\rho^2(1+C_s)(C_a+\rho^2C_s)}{2\lambda(1-\rho)(1+\rho^2C_s)}$

Table 1: Queue types and assumptions

second letter describes service time, and the number tells how many queues are in the system. The simplest model, $M/M/1$ assumes both inter-arrival and service time to be approximated by a Poisson process. $M/D/1$ model makes the service time to be constant, and $M/G/1$ queue assumes service time to follow a general distribution. The most general model we have considered is $G/G/1$. It assumes both the service-time and inter-arrival time as any given distributed random variables. We summarize and compare each queueing type in Table 1.

3.3 Modeling Consensus Performance

To model consensus performance, we are interested in estimating the average latency of a consensus round as perceived by the client. For such latency estimates, we consider the parameters outlined in Table 2. The average latency is comprised of a few different components: round's queue waiting time w_Q, round's service time t_s, and network delays D_L and D_Q:

$$Latency = w_Q + t_s + D_L + D_Q$$

For Paxos, the network delays consist of a round-trip time (RTT) between the client and the leader D_L, and between the leader and some follower that sends the message forming a quorum of replies at the leader D_Q. For the network delays, we assume only the time-in-transit for messages, since the time to clear NIC is accounted in our service time computations. To calculate D_Q in Paxos, we need to consider the quorum size Q of the deployment. For a cluster with N nodes, the quorum size is $Q = \lfloor \frac{N}{2} \rfloor + 1$, making a self-voting leader wait for $Q - 1$ follower messages before reaching a majority quorum. The RTT for this $(Q-1)^{\text{th}}$ follower reply is D_Q. In LAN we assume the RTTs between all nodes to be drawn from the same Normal distribution, therefore we use a Monte Carlo method approximation of k-order-statistics [14] to compute the RTT for $(Q-1)^{\text{th}}$ reply. In WAN, however, the RTTs between different nodes may be significantly different, therefore we pick the $(Q-1)^{\text{th}}$ smallest RTT between the leader and its followers.

Round service time t_s is a measure of how long it takes the leader to process all messages for a given round. For

Table 2: Model parameters

N	Number of nodes participating in a Paxos phase
Q	Quorum size. For a majority quorum $Q = \lfloor \frac{N}{2} \rfloor + 1$
D_L	RTT between client and the leader node.
D_Q	RTT between the leader and $(Q-1)^{th}$ follower
b	Network bandwidth available at the node
s_m	Message size
t_i	Processing time for incoming message
t_o	Processing time for outgoing message
t_s	Service time. For Paxos: $t_s = 2t_o + Nt_i + \frac{2Ns_m}{b}$
μ	Max throughput or service rate. $\mu = \frac{1}{t_s}$
λ	Workload throughput or arrival rate in rounds per second
ρ	Queue utilization. For $M/D/1$ queue: $\rho = \frac{\lambda}{\mu}$
w_Q	Queue wait time for a round. For $M/D/1$ queue: $w_Q = \frac{\rho}{2\mu(1-\rho)}$

simplicity, we assume that each message, incoming and outgoing, needs to be processed by both the NIC and CPU. The round's service time t_s is then a sum of t_{NIC} and t_{CPU}: $t_s = t_{NIC} + t_{CPU}$. We compute t_{NIC} as the time required to push all round's messages of size s_m through the network of some bandwidth b: $t_{NIC} = \frac{2Ns_m}{b}$. CPU portion of the service time is estimated as the sum of costs for processing incoming messages t_i and outgoing messages t_o. For Paxos, each phase-2 round requires the leader to receive a message from the client (t_i), broadcast one outgoing message (t_o because the CPU serializes the broadcast message once), receive $N-1$ messages from the followers ($(N-1) * t_i$), and reply back to the client (t_o). As a result, we have $t_{CPU} = 2t_o + N * t_i$.

Note that only service time impacts the maximum throughput of the system, since it alone governs how much of the processing resources are consumed for each round. The protocol reaches its peak throughput when it fully saturates the queue and leaves no unused resources in it. The maximum throughput is the reciprocal of service time:

$$\mu = \frac{1}{t_s}$$

Finally, we estimate the queueing costs w_Q by computing average queue wait time.

To select the most appropriate queue approximation, we used all four queuing models described earlier to estimate

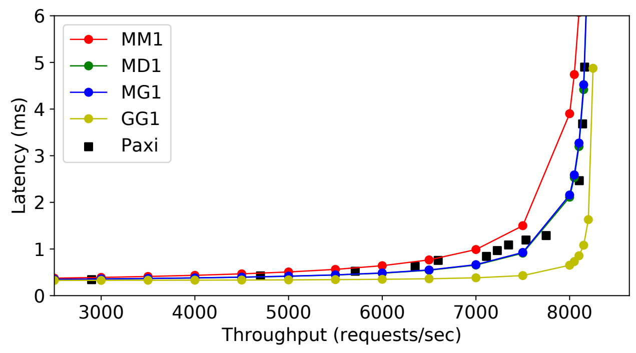


Figure 4: Comparison of different queueing models to a reference implementation in Paxi framework

the performance of Paxos protocol in LAN. We used empirically obtained values for all relevant parameters, such as network RTTs, serialization/deserialization costs, etc. Figure 4 shows the comparison between the models and a real Paxos implementation in Paxi. Based on our findings, $M/D/1$ and $M/G/1$ models perform nearly identical and most similar to our reference Paxos implementation. Since $M/D/1$ model is simpler, we use it for all our further analysis.

3.4 Expanding Models Beyond Paxos and LANs

The other protocols, albeit being more complicated, share the same modeling components with the Paxos described above. Thus, we rely on the same queuing model and round latency computation principles for them as well.

Multi-leader protocols add a few additional parameters to consider. In such protocols a node can both lead the round and participate in concurrent rounds as a follower, therefore we account for the messages processed as a follower in addition to the messages processed as round's leader to estimate queue waiting time. The total number of requests coming to the system is spread out evenly (we assume uniform workload at each leader for simplicity) across all leaders. This allows us to compute per-leader latency based on each leader's processing queue.

The performance of leaderless protocols, such as EPaxos, varies with respect to the command conflict ratios. A conflict typically arises when two or more replicas try to run a command against the same conflict domain (e.g., the same key) concurrently. The conflict must be resolved by the protocol, which leads to extra message exchanges. Therefore, we introduce a conflict probability parameter c for EPaxos to compute different latencies for conflicting and non-conflicting commands. The overall average latency of a round accounts for the ratio of conflicting to non-conflicting commands as follows: $Latency = c(Latency_{conflict}) + (1 - c)(Latency_{nonconflict})$.

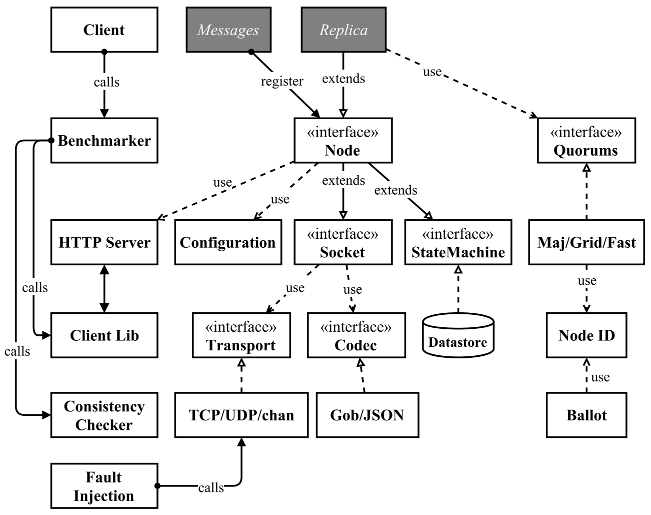


Figure 5: Paxi modules usage where user implements Messages and Replica type (shaded)

WAN modeling also requires changes over the LAN model. In particular, communication latency between nodes in WAN cannot be drawn from the same distribution, since the datacenters are not uniformly distanced from each other. For example, in a 3-node Paxos configuration with replicas in Eastern U.S., Ireland and Japan, the communication latency between US and Ireland is significantly smaller than that of Ireland and Japan or US and Japan. For that reason, our WAN modeling no longer assumes the same Normal distribution of latency for all nodes, and instead we use a different distribution for communication between every pair of datacenters.

4 PAXI FRAMEWORK

In order to provide empirical comparisons of different protocols in the same platform under the same conditions, we developed a prototyping framework for coordination/replication protocols called Paxi. Leveraging our observation that strongly consistent replication protocols share common parts, Paxi provides implementations for these shared components. Each component resides in its own loosely coupled module and exposes a well-designed API for interaction across modules. Modules can be extended or replaced easily, provided that a new version a module follows the interface. We show the architectural overview of Paxi framework in Figure 5. The developer can easily prototype a distributed coordination/replication protocol by filling in the missing components shown as the shaded blocks. Often the engineers only need to specify message structures and write the replica code to handle client requests.

4.1 Components

Configurations. Paxi is a highly configurable framework both for the nodes and benchmarks. A configuration in Paxi provides various vital information about the protocol under examination and its environment: list of peers with their reachable addresses, quorum configurations, buffer sizes, replication factor, message serialization and networking parameters, and other configurable settings. The framework allows to manage configuration in two distinct ways: via a JSON file distributed to every node, or via a central master that distributes the configuration parameters to all nodes.

Quorum systems. A quorum system is a key abstraction for ensuring consistency in fault-tolerant distributed computing. A wide variety of distributed coordination algorithms rely on quorum systems. Typical examples include consensus algorithms, atomic storage, mutual exclusion, and replicated databases. Paxi implements and provides multiple types of quorum systems, like simple majority, fast quorum, grid quorum, flexible grid and group quorums. The quorum system module only needs two simple interfaces, *ack()* and quorum *satisfied()*. By offering different types of quorum systems out of the box, Paxi enables users to easily probe the design space without changing their code.

Networking. When designing Paxi framework, we refrained from any blocking primitives such as remote procedure calls (RPC), and implemented a simple message passing model that allows us to express any algorithmic logic as a set of event handlers [36, 38]. The networking module transparently handles message encoding, decoding, and transmission with a simple Send(), Recv(), Broadcast() and Multicast() interface. The transport layer instantiates TCP, UDP, or Go channel for communication without any changes to the caller from an upper layer. Paxi supports both TCP and UDP to eliminate to eliminate any bias for algorithms that benefit from different transport protocols. For example, the single leader approach may benefit from TCP as messages are reliably ordered from leader to followers. Whereas conflict-free updates in small messages gain nothing from ordered delivery and pay the latency penalty in congestion control. Such system might perform better on UDP. Paxi also implements the intra-process communication transport layer by Go channels for a cluster simulation, where all nodes run concurrently within a single process. The simulation mode simplifies the debugging of Paxi protocols since it avoids a cluster deployment step.

Data store. Many protocols separate the protocol-level outputs (e.g. operation commits) from the application state outputs (e.g operation execution result) for versatility and performance reasons. Therefore, evaluating for either one is insufficient. Paxi can be used to measure both protocol and application state performance. To this end, our framework comes with an in-memory multi-version key-value datastore that is private to every node. The datastore is used as a deterministic state machine abstraction commonly used by coordination protocols. Any other data model can be used for this purpose as long as Paxi node can query current state, submit state transform operation and generate directed acyclic graph (DAG) of past states.

Parameter	Default Value	Description
T	60	Run for T seconds
N	0	Run for N operations (if N>0)
K	1000	Total number of keys
W	0.5	Write ratio
Concurrency	1	Number of concurrent clients
LinearizabilityCheck	true	Check linearizability at the end of benchmark
Distribution	"uniform"	Name of distribution used for key generation include uniform, normal and zipfian
Min	0	Random: minimum key number
Conflicts	100	Random: percentage of conflicting keys
Mu	0	Normal: Mean
Sigma	60	Normal: Standard Deviation
Move	false	Normal: Moving average (mu)
Speed	500	Normal: Moving speed in milliseconds
Zipfian_s	2	Zipfian: s parameter
Zipfian_v	1	Zipfian: v parameter

Table 3: Benchmark parameters

RESTful client. The Paxi client library uses a RESTful API to interact with any system node for read and write requests. This allows users to run any benchmark (e.g. YCSB [11]) or testing tools (e.g. Jepsen [22]) against their implementation in Paxi without porting the client library to other programming languages.

4.2 Paxi Benchmark Components

Benchmarker. The benchmarker component of Paxi serves as the core of our benchmarking capabilities. The benchmarker both generates tunable workloads with rich features including access locality and dynamicity and collects performance data for these workloads. Similar to YCSB [11], Paxi provides a simple read/write interface to interact with the client library. The workload generator reads the configuration to load the workload definition parameters, as summarized in Table 3.

Benchmark component can generate a variety of workloads by tuning the read-to-write ratios, creating hot objects, conflicting objects and locality of access. The locality characteristic in workload is especially important in WAN distributed protocols as each region has a set of keys it is more likely to access. In order to simulate workloads with tunable access locality patterns across regions, Paxi uses a normal distribution to control the probability of generating a request on each key, and denotes a pool of K common keys with the probability function of each region, as shown in Figure 6. In other words, Paxi introduces locality to the evaluation by drawing the conflicting keys from a Normal

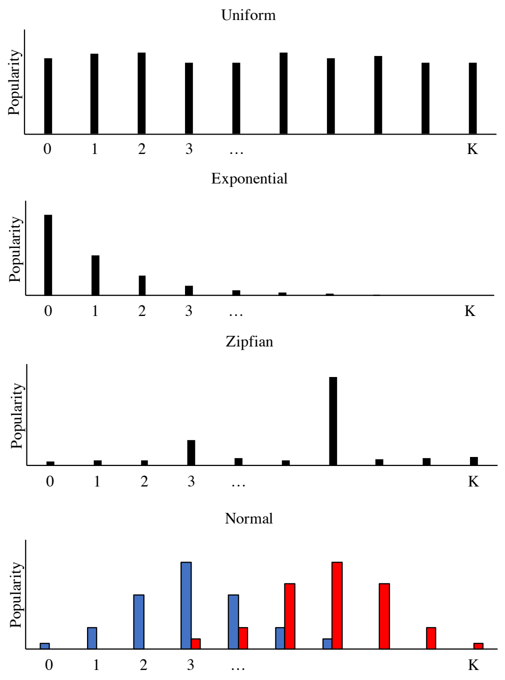


Figure 6: Probability distributions with total number of data records equal to K

distribution $\mathcal{N}(\mu, \sigma^2)$, where μ can be varied for different regions to control the locality, and σ is shared between regions. The locality can be visualized as the non-overlapping area under the probability density functions in Figure 6.

Paxi benchmarker is capable of testing and evaluating four different aspects of coordination/replication protocols behavior: performance, scalability, availability, and consistency.

Performance. Paxi measures performance via the latency and throughput metrics. The latency of every individual request is stored and output to a file for future analysis. Since it is important to show how a protocol perform in terms of tail latency under stress, Paxi supports this by increasing the benchmark throughput (via increasing the concurrency level of the workload generator) until the system is saturated and throughput stops increasing or latency starts to climb. The user may conduct this benchmark tier with different workloads to understand how a protocol performs or use the same workload with increasing throughput to find a bottleneck of the protocol.

Scalability. One of the most desirable properties for cloud native protocols is the elastic scalability: when the application grows/shrinks, the underlining protocol should be able to adapt to load by expanding/reducing number of servers. In Paxi, we support benchmarking scalability by adding more nodes into system configuration and by increasing the size of dataset (K).

Availability. High availability is an indispensable requirement for distributed protocols, as they are expected to maintain progress under a reasonable number of failures. While testing for availability seems straightforward, it requires laborious manual work to simulate all combinations of failures. Many failure types are hard to produce in an uncontrolled runtime environments without utilizing third party tools specific to their operating systems. Typical examples include asymmetric network partition, out of order messages and random message drop/delay, to name only a few. Several projects have automated fault injection procedures, but with limitations. For instance, Jepsen [22] issues "tc" (traffic control) commands to manipulate network packets on every node, but can only run on Linux systems. ChaosMonkey [32] is a resiliency tool that only randomly terminates virtual machine instances and containers. Paxi, being a prototyping framework, can make it easy to simulate any node or network failure. We provide four special commands in the Paxi client library and realize those in the networking modules:

- Crash(t) will “freeze” the node for t seconds.
- Drop(i, j, t) function drops every message send from node i to node j.
- Slow(i, j, t) function delays messages for a random period.
- Flaky(i, j, t) function drops messages by chance.

Consistency. For the consistency checker component of Paxi, we implement the simple offline read/write linearizability checker from the Facebook TAO system [28]. Our linearizability checker takes a list of all the operations per record sorted by invocation time as an input. The output of the checker is a list of all anomalous reads, i.e., read operations that returned results they would not be able to in a linearizable system. Our checker maintains a graph whose vertices are read or write operations, and edges are constraints. It checks for cycles as it adds operations to the graph. If the graph is acyclic, then there exists at least one total order over the operations with all constraints satisfied. Else linearizability violation is reported.

Unlike the linearizability checker, our consensus checker validates whether the consensus for every state transition has been reached among the nodes in a replicated state machine. External, client-observed, linearizability can be reached without having a consensus among the state machines, however, satisfying consensus is vital for consensus algorithms, such as Paxos. To test for consensus, Paxi includes a multi-version datastore as the state machine. We implement a special command in client library to collect entire history of some data record H^r from every system node, then verify if all history H^r_i from node i shares a common prefix.

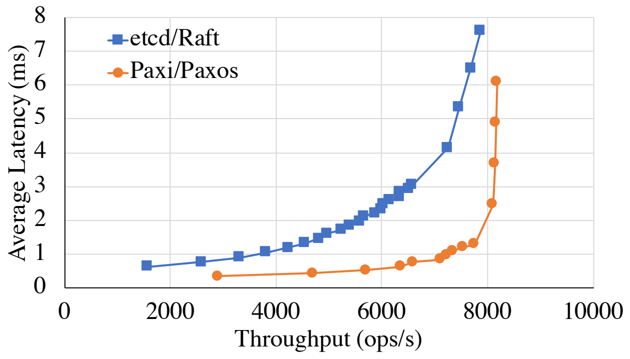


Figure 7: Single leader consensus protocol implemented in Paxi versus etcd

5 EVALUATION OF THE PROTOCOLS

We perform protocol evaluations using both simulated experiments with our model and deployed experiments with Paxi framework in LANs and WANs. In Paxi, we carry out all experiments on AWS EC2 [5] m5.large instances with 2 vCPUs and 8 GB of RAM. For WAN evaluations, we use AWS datacenter topology with the N.Virginia (VA), Ohio (OH), California (CA), Ireland (IR) and Japan (JP) regions. Our modeling effort is calibrated to CPU speed of the m5.large instances and uses the communication delays corresponding to latencies within an AWS region and across regions.

In WPaxos, we limit the number of nodes that can act as leaders to just one per region; in a 9-node 3-region cluster WPaxos will have only 3 leaders. This gives WPaxos similar and comparable deployment to WanKeeper that is restricted by its design to have just a single leader in each region. In EPaxos, every node can become an opportunistic leader. Additionally, EPaxos model penalizes the message processing to account for extra resources required to compute dependencies and resolve conflicts. For all protocols we assume full-replication scheme in which a leader node replicates the commands to all other nodes.

5.1 Paxi Performance

The Paxi framework serves the primary goal of comparing many different consensus and replication protocols against each other under the same framework with the same implementation conditions. However, to show that our Paxi implementation of protocols are representative of Paxos variant implementations used in other real-world production grade systems, we compared the Paxos protocol implemented in Paxi against Raft [33] implemented in etcd [15]. The etcd project provides a standalone sample code in Go[1] that uses Raft and exposes a simple REST API for a key-value store, allowing us to directly run Paxi benchmark against etcd without any changes.

Without considering reconfiguration and recovery differences, Paxos and Raft are essentially the same protocol with a single stable leader driving the command replication. As a result, they should exhibit similar performance in the normal case. We ran each system with 9 replicas within the same availability zone. For a fair comparison with Paxi, we disabled persistent logging and snapshots and increased the maximum number of inflight messages to 10,000 in etcd. The client request is replied only after the request is committed in Raft. In Figure 7, we show that both systems converge to similar maximum throughput around 8000 operations per second due to the single leader bottleneck, but Paxi exhibits lower latency when the system is not saturated. The latency difference is likely due to etcdâĂŹs use of http for inter-node communication instead of TCP sockets and differences in message serialization.

5.2 Protocol Comparisons in LANs

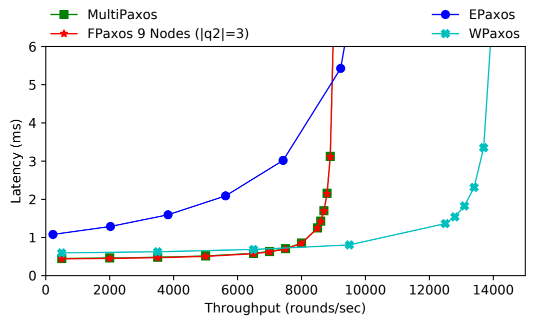
We perform a set of experiments studying the performance of the consensus protocols in LANs. Figure 8 shows the results obtained from our model for LANs. We present the Paxi evaluation in Figure 9 using uniformly random workload against Paxi's internal key-value store on 1000 objects with 50% read operations. Both figures show how latency of protocols change as the throughput increases.

Single leader bottleneck. Both the model and Paxi evaluations point to the scalability limitation of the single leader solutions. Several papers [20, 27, 29, 39] observed—but failed to analyze further— that the single leader becomes a bottleneck in Paxos protocol, having to do with sending/receiving N messages and the CPU utilization at the leader. The bottleneck is due to the leader getting overwhelmed with the messages it needs to process for each round. For example, from our modeling effort, we estimate a Paxos leader to handle N incoming messages, one outgoing message and one outgoing broadcast[2], for a total of $N + 2$ messages per round. At the same time, the follower nodes only process 2 messages per round, assuming phase-3 is piggybacked to the next phase-2 message. For a cluster of 9 nodes, this translates in 11 messages per round at the leader against just 2 messages at the replicas, making the leader the bottleneck.

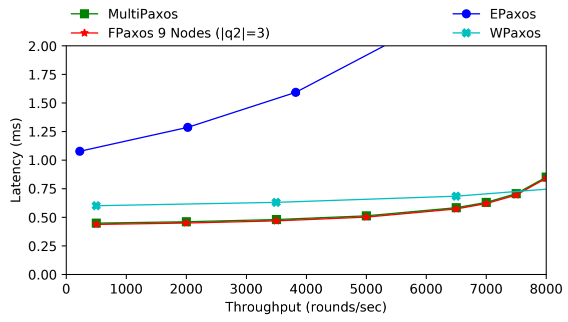
Multi-leader protocols, such as WPaxos and WanKeeper perform better than single leader protocols. Their advantage comes from the ability to process commands for independent objects in parallel at different nodes. While multi leader

[1]https://github.com/etcd-io/etcd/tree/master/contrib/raftexample

[2]For the broadcast, the CPU is involved for serialization just once, and then the message is send to the other nodes individually by the NIC (amounting to N-1 transmission). Since NIC is much faster than the CPU processing, the NIC cost becomes negligible for small messages.



(a) Max throughput



(b) Latency at lower throughput

Figure 8: Modeled performance in LANs

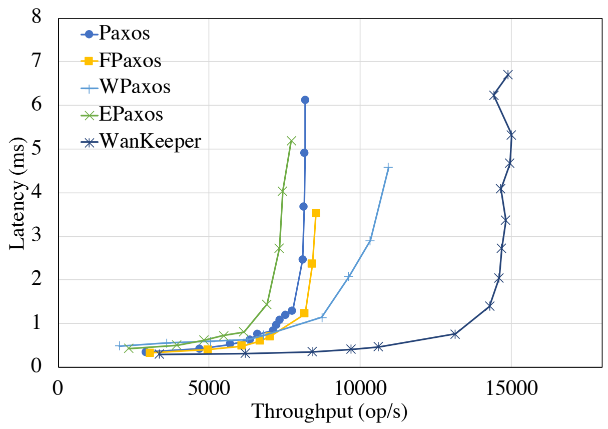


Figure 9: Experimental performance in LAN

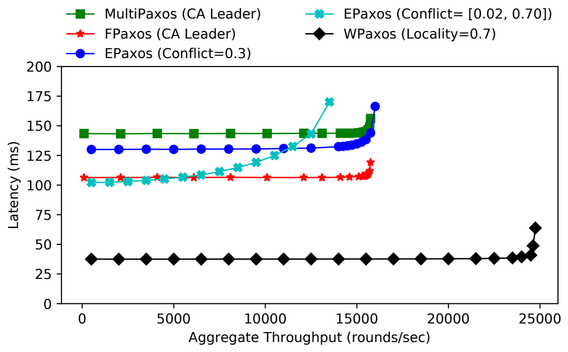


Figure 10: Modeled performance in WANs

solutions can take advantage of the capacity left unused at the single-leader protocol replicas, they still suffer from the relatively high number of messages to process, so they don't scale linearly: the 3-leader WPaxos does not perform 3 times better than Paxos. Our model showed a roughly 55% improvement in maximum throughput in WPaxos, which is consistent with our experimental observations in Paxi.

The figures also show that the 3-leader WanKeeper performs better than the 3-leader WPaxos. By taking a hierarchical approach, WanKeeper reduces the number of messages each leader processes, alleviating the leader bottleneck further.

Leaderless solutions suffer from conflict. EPaxos is an example of a leaderless consensus, where each node can act as an opportunistic leader for commands, as long as there are no collisions between the commands from different nodes. Command conflicts present a big problem for such opportunistic approaches. The ability to identify and resolve conflicts increases the message size and processing capacity needed for messages. The conflict resolution mechanism also requires an additional phase, making conflicting rounds default to a two-phase Paxos implementation. These result in EPaxos performing the worst in Paxi LAN experiments. It is worth mentioning that EPaxos shows better throughput (but not latency) than Paxos in our model even with 100% conflict, since it does not have a single-leader bottleneck problem. However, when we add message processing penalty to account for extra weight of finding and resolving conflict, EPaxos' performance degrades greatly.

Small flexible quorums benefit. Our model results show a modest average latency improvement of just 0.03 ms due to using single-leader flexible quorums solution (FPaxos). Our Paxi evaluation shows a slightly bigger improvement for going from Paxos to FPaxos.

5.3 Protocol Comparison in WANs

In this set of experiments, we compare protocols deployed across the WANs. WAN deployments present many new challenges for consensus and replication protocols originating from large and non-uniform latencies between datacenters and nodes: the inter-datacenter latency can vary from a few to a few hundred milliseconds.

In Figure 10 we show the modeled throughput and latency results for different consensus algorithms.

Unlike LAN models, wide area network models differ greatly in the average latency, with more than a 100 ms difference in latency between the slowest (Paxos) and the fastest (WPaxos) protocols. Flexible quorums make a great difference in latency in this environment – they allow WPaxos to commit many commands with near local latency, and reduce the overall quorum wait time for FPaxos.

To combat the adversarial effects of WAN, many protocols try to optimize for various common aspects of operation in such environment: some assume the objects over which a consensus is needed get rarely accessed from many places at once, while others may go even further and assume a strict placement of the objects in certain regions for better locality of access. When these assumptions break, algorithms' performance often degrades. We designed our experiments to test these conflict and locality assumptions.

Conflict Experiments. We define conflict as commands accessing the same object from different region. To study the protocol performance under the conflict workload, we create one "hot" conflicting key that will be accessed by all clients. We control the ratio of conflicting requests and make every conflicting request perform an operation against the designated conflict objects. For instance, 20% conflict mean that 20% of requests issues by the benchmarker clients target the same object.

The results of our conflict experiments, shown in Figure 11, reveal the following observations:

(1) The protocols that does not tolerate entire region failure (WPaxos $f_z = 0$, WanKeeper, VPaxos) exhibit the same performance in every location. This is because when $f_z = 0$, the non-interfering commands are able to commit by a quorum in the same region. The interfering command is forwarded to the object's current leader region.

(2) When the protocol embraces a leader/owner concept, the leader's region of the conflicting object is at an advantage and experience optimal latency for its local quorum. In this case, the Ohio region is the leader of conflicting object, and thus have a low steady latency. On the contrary, EPaxos, a leaderless approach, experiences latency due to interfering commands even in the Ohio region.

(3) Among the protocols which can tolerate entire region failure, including Paxos, EPaxos, and WPaxos fz = 1, WPaxos performs best until 100% interfering commands where it starts to provide the same latency as Paxos.

(4) Unlike other protocols, EPaxos average latency is a non-linear function of conflicting ratio. This is because even with low conflict rate like 20%, the previous conflicting command may not have been committed yet when the new requests arrive, leading to more rounds of RTT delays to resolve the conflict. The situation gets worse when the region is far from other regions, such as California region in our experiment.

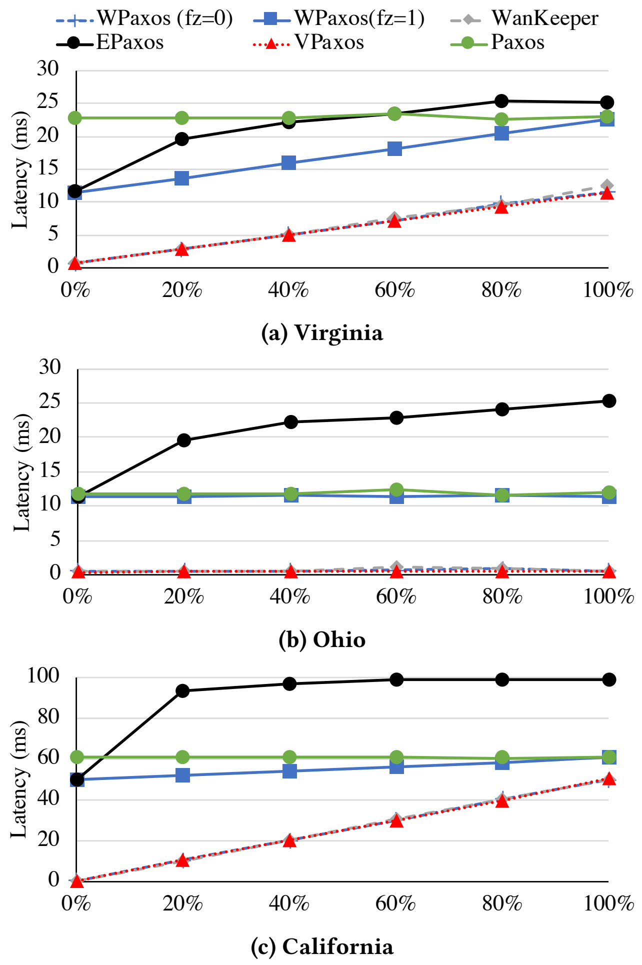


Figure 11: Comparison of protocols under a conflict workload

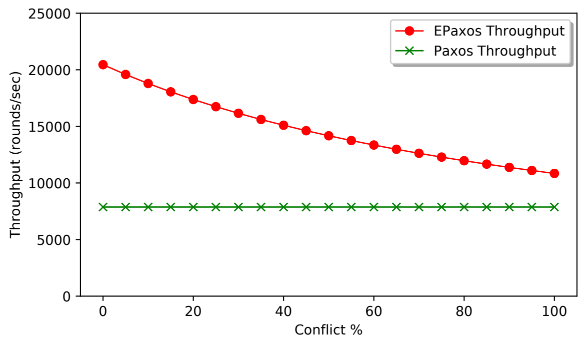
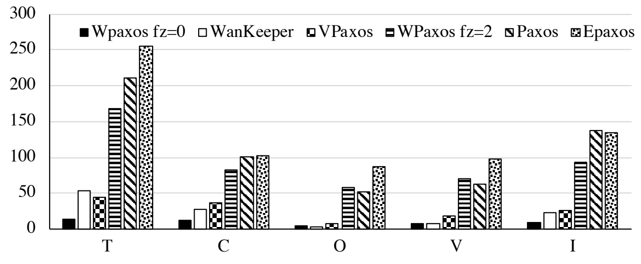
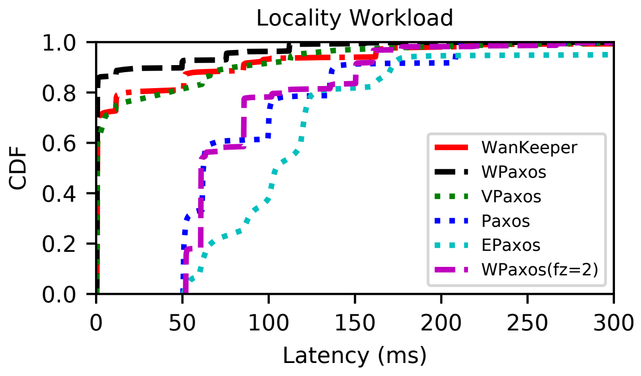


Figure 12: Model of EPaxos maximum throughput in 5-nodes/regions deployment

Our model also suggests the maximum throughput of EPaxos is severely affected by the conflict ratio, with as much



(a) Average latency per region



(b) Latency distribution

Figure 13: WPaxos, WanKeeper and Vertical Paxos in WAN with locality workload

as 40% degradation in capacity between no conflict and full conflict scenarios, as illustrated in Figure 12.

Locality Experiments. For these experiments, we study the performance of protocols that take advantage of locality (WPaxos, WanKeeper, and our augmented version of Vertical Paxos) across a WAN with a locality workload. We use a locality workload with object access locality controlled by a normal distribution, as described earlier. We start the experiment by initially placing all objects in the Ohio region and then running the locality workload for 60 seconds. All three protocols are deployed with fault-tolerance level $f_z = 0$ and the simple three-consecutive access policy to adapt to optimal performance. Figure 13a compares per-region latency, while Figure 13b shows a CDF for operation latencies.

In WanKeeper, the Ohio region is the higher level region that will keep the tokens for any objects shares access from another region. Thus, Ohio shows the best average latency close to local area RTT, at the cost of suffering at the other two regions. WPaxos and VPaxos are more balanced and share very similar performance in this deployment. This is because when stabilized, both protocols balance the number objects in each region in the same way, as confirmed by

	Name	Protocols
L	Leaders	EPaxos. WPaxos
c	Conflicts	Generalized Paxos, EPaxos
Q	Quorum	FPaxos, WPaxos
l	Locality	VPaxos, WPaxos, WanKeeper

Table 4: Parameters explored

almost identical CDFs in Figure 6.4. When we look at all requests globally, WanKeeper experience more WAN latencies than WPaxos and VPaxos.

6 DISCUSSION

In this section, we examine the evaluation results from Section 5 and distill those performance results into simple throughput and latency formulas for uniformly distributed workloads. These formulas present a simple unified theory of strongly-consistent replication throughput in Section 6.1, and latency in Section 6.2. Finally, in Section 6.3, we demonstrate how these formulas allow us to perform back-of-the-envelope performance forecasting of the protocols.

6.1 Load and Capacity

The capacity of the system $Cap(S)$ is the highest request processing rate that system S can handle. As we have observed from Paxi experiments, the capacity of a given protocol is determined by the busiest node in the system with load $\mathcal{L}$ [?], such that

$$Cap(S) = 1/\mathcal{L}(S). \tag{1}$$

Definition 6.1. **Load** of the system $\mathcal{L}(S)$ is the minimum number of operations invoked on the busiest node for every request on average, where an operation is the work required to handle round-trip communication between any two nodes. For example, for every quorum access, the leader must handle Q number of outgoing and incoming messages, which correspond to a total of Q operations. Whereas in single-leader protocols, the busiest node is typically the leader, multi-leader algorithms have more than one busiest node. In such protocols, nodes that have the leader capabilities tend to be under greater load than others.

$$\mathcal{L}(S) = \frac{1}{L}(1+c)(Q-1) + (1-\frac{1}{L})(1+c) \tag{2}$$

$$= \frac{(1+c)(Q+L-2)}{L} \tag{3}$$

where $0 \leq c \leq 1$ is probability of conflicting operations, Q is the quorum size chosen by leader, and L is the number of operation leaders. The derivation is as follows. There is a $1/L$ chance the node is the leader of a request, which induces one

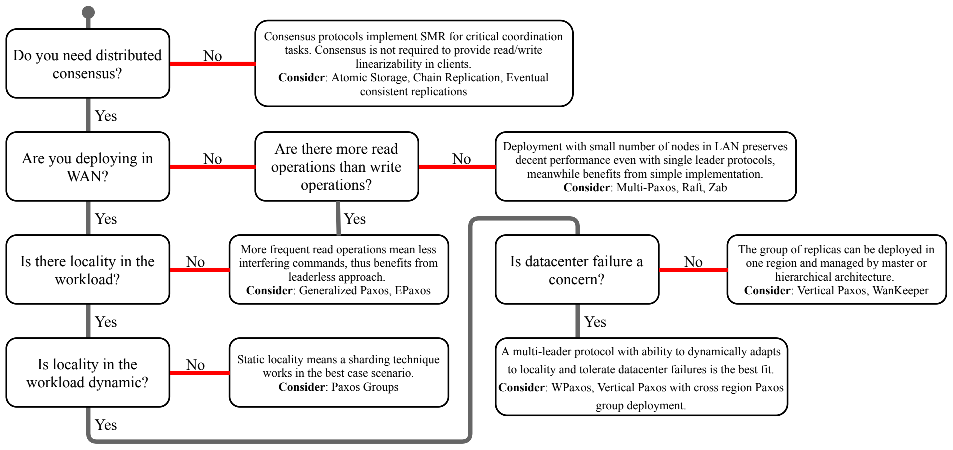

Do you need distributed consensus?
No
Consensus protocols implement SMR for critical coordination tasks. Consensus is not required to provide read/write linearizability in clients. **Consider**: Atomic Storage, Chain Replication, Eventual consistent replications
Yes
Are you deploying in WAN?
No
Are there more read operations than write operations?
No
Deployment with small number of nodes in LAN preserves decent performance even with single leader protocols, meanwhile benefits from simple implementation. **Consider**: Multi-Paxos, Raft, Zab
Yes
Yes
Is there locality in the workload?
No
More frequent read operations mean less interfering commands, thus benefits from leaderless approach. **Consider**: Generalized Paxos, EPaxos
Is datacenter failure a concern?
No
The group of replicas can be deployed in one region and managed by master or hierarchical architecture. **Consider**: Vertical Paxos, WanKeeper
Yes
Yes
A multi-leader protocol with ability to dynamically adapts to locality and tolerate datacenter failures is the best fit. **Consider**: WPaxos, Vertical Paxos with cross region Paxos group deployment.
Is locality in the workload dynamic?
No
Static locality means a sharding technique works in the best case scenario. **Consider**: Paxos Groups
Yes

Figure 14: Flowchart for identifying the suitable consensus protocol

round of quorum access with $Q-1$ communication, plus extra round of quorum access if there is conflict. The probability of the node being a follower is $1-1/L$, where it only handles one received message in the best case. From the simplified form, it is easy to see that the protocols that utilize more leaders reduce the load (and hence increase capacity) because the user requests are shared between multiple leaders. On the other hand, this also increases the chance of extra round to resolve any conflicts between leaders. Equation 3 uses thrifty optimization where leader only communicates with minimum number of nodes to reach the quorum of size Q. In the general case, however, the leader communicates with all $N-1$ followers, making $Q = N-1$ for the purpose of this equation.

In the below equations, we present the simplified form of load for three protocols, and calculate the result for $N = 9$ nodes. The protocols perform better as the load decreases.

$$\mathcal{L}(\text{Paxos}) = \lfloor \frac{N}{2} \rfloor \qquad = 4 \tag{4}$$

$$\mathcal{L}(\text{EPaxos}) = (1+c)(\lfloor \frac{N}{2} \rfloor + N - 1)/N \qquad = \frac{4}{3}(1+c) \tag{5}$$

$$\mathcal{L}(\text{WPaxos}) = (\frac{N}{L} + L - 2)/L \qquad = \frac{4}{3} \tag{6}$$

In the single leader Paxos protocol (Equation 4) with N nodes, $c = 0$ as the conflicting operation is serialized by the single leader, and $L = 1$, and quorum size $Q = \lfloor N/2 \rfloor + 1$. In contrast, EPaxos (Equation 5) uses every node as an opportunistic leader, and uses $L = N$. WPaxos (Equation 6) utilizes a flexible grid quorum system, such that every leader only accesses its own phase-2 quorum with size N/L. In a 3×3 grid, the load of WPaxos is only 4/3, giving it the smallest load (and as a result the highest capacity) among the three consensus protocols.

6.2 Latency

The expected latency of a protocol in WAN is determined by the location, minimum quorum latency D_Q, and the *locality* l of the requests.

$$Latency(S) = (1+c) * ((1-l) * (D_L + D_Q) + l * D_Q) \tag{7}$$

where D_L is the distance from where the request is generated and the operation leader. When a request is local with probability l, it only requires time of the quorum access with closest neighbors D_Q. For non-local requests occurred with probability of $1-l$, a round trip of distance to leader D_L also contributes to the average latency. For EPaxos $l = 1$ but c is workload specific, in contrast for the other consensus protocols we study, we have $c = 0$ and l is workload specific.

6.3 Comparing the Protocols

By using the two formulas 3 and 7 we present in the previous subsections, we discuss how these protocols compare with each other and how we can provide back-of-the-envelope performance forecasting of these protocols.

EPaxos and Generalized Paxos try to make single leader Paxos more scalable by extending the leadership to every replica therefore sharing the load; this increases L to its maximum value of all nodes in the system and reduces $\mathcal{L}(S)$. But it comes with a complication: any conflicting commands from two replicas require extra round of quorum acknowledgement for order resolution before the requests can be executed and replied. This extra round puts more load to the system, reducing throughput and increasing latency. Our evaluations show that the problem becomes even worse in WANs: since requests take much longer time to finish in WANs, that also contributes to an increase in the probability of contention. In the worst case, even with 25% of conflicting requests, system can experience $c = 100\%$ load.

Flexible-Paxos and WPaxos benefits from flexible quorums such that both Q and D_Q are reduced for phase-2.

The three WAN protocols we evaluated in this work exploit the locality l in the workload to optimize latency for wide area. When the locality is static, and an optimal policy is used for placing the data close to most requests, these protocols will experience the same WAN latency.

In Table 4, we show the parameters each protocol aims to explore. Given that each protocol emphasizes a couple of these parameters and trades them off with others, there is no one protocol that fits all needs/expectations. Our results and formulas are also useful for deciding which category of Paxos protocols would be optimal under given deployment conditions. In Figure 14, we give a flowchart to serve as a guideline to identify which consensus protocol would be suitable for a given deployment environment.

7 CONCLUDING REMARKS

We presented a two pronged approach to analyze the strongly-consistent replication protocols. We distilled the throughput and latency performance to simple formulas that generalize over Paxos protocols, uniting them, as well as emphasizing the different design decisions and tradeoffs they take.

We anticipate that the simple exposition and analysis we provide will lead the way to the development of new protocols, especially WAN coordination protocols. The unbalanced topology with respect to obtaining a quorum causes complications in WAN deployments, and achieving good locality as well as load balancing remains an open problem for efficient strongly-consistent WAN replication. In addition, as part of future work, we aim to extend our analytical model to cover replication protocols with relaxed consistency guarantees, such as bounded-consistency and session consistency.

ACKNOWLEDGMENTS

This project is in part sponsored by the National Science Foundation (NSF) under award number CNS-1527629 and XPS-1533870.

REFERENCES

[1] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2017. Multileader WAN Paxos: Ruling the Archipelago with Fast Consensus. *arXiv preprint arXiv:1703.08905* (2017).

[2] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Bekir Oguz Turkkan, and Tevfik Kosar. 2017. Efficient Distributed Coordination at WAN-scale. In *Distributed Computing Systems (ICDCS), 2017 37th International Conference on*. IEEE.

[3] Arnold O. Allen. 2014. *Probability, statistics, and queueing theory.* Academic Press.

[4] Deniz Altinbuken and Emin Gun Sirer. 2012. *Commodifying replicated state machines with openreplica.* Technical Report.

[5] Amazon Inc. 2008. Elastic Compute Cloud.

[6] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. 2012. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment* 5, 8 (2012), 776–787.

[7] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11).*

[8] Jim Campigli and Yeturu Aahlad. 2018. The Distributed Coordination Engine (DConE). *WANDisco, Inc* (2018).

[9] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2018. *FleetDB: Follow-the-workload Data Migration for Globe-Spanning Databases.* Technical Report. https://cse.buffalo.edu/tech-reports/2018-02.pdf

[10] Clustrix. 2017. A New Approach to Scale-out RDBMS: Massive Transactional Scale for an Internet-Connected World. https://www.clustrix.com/wp-content/uploads/2017/01/Whitepaper-ANewApproachtoScaleOutRDBMS.pdf. (2017).

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10).* ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[12] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. 2012. Spanner: Google's globally-distributed database. *Proceedings of OSDI* (2012).

[13] Citus Data. 2016. Master-less Distributed Queue with PG Paxos. https://www.citusdata.com/blog/2016/04/13/masterless-distributed-queue/. (2016).

[14] Herbert Aron David and Haikady Navada Nagaraja. 2004. Order statistics. *Encyclopedia of Statistical Sciences* 9 (2004).

[15] etcd [n. d.]. etcd: Distributed reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/.

[16] Matt Freels. 2018. FaunaDB: An Architectural Overview. https://fauna-assets.s3.amazonaws.com/public/FaunaDB-Technical-Whitepaper.pdf

[17] Google. 2018. The Go Programming Language. https://golang.org/ https://golang.org/.

[18] Heroku. 2018. Doozerd. https://github.com/ha/doozerd.

[19] Ezra N Hoch, Yaniv Ben-Yehuda, Noam Lewis, and Avi Vigder. 2017. Bizur: A key-value consensus algorithm for scalable file-systems. *arXiv preprint arXiv:1702.04242* (2017).

[20] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696* (2016).

[21] F. Junqueira, B. Reed, and M. Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN).* IEEE, 245–256.

[22] Kyle Kingsbury. 2017. Jepsen Tests. https://jepsen.io/ https://jepsen.io/.

[23] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems.* ACM, 113–126.

[24] Cockroach Labs. 2018. CockroachDB: The SQL database for global cloud services. https://www.cockroachlabs.com/docs/stable/architecture/overview.html.

[25] L. Lamport. 2001. Paxos made simple. *ACM SIGACT News* 32, 4 (2001), 18–25.

[26] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 312–313.

[27] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say {NO} to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 467–483.

[28] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 295–310.

[29] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. 2008. Mencius: building efficient replicated state machines for WANs. In *OSDI*, Vol. 8. 369–384.

[30] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 358–372.

[31] Neo4j. 2012. Neo4j - The WorldâĂŹs Leading Graph Database. http://neo4j.org/

[32] Netflix. 2017. Chaos Monkey. https://github.com/Netflix/chaosmonkey https://github.com/Netflix/chaosmonkey.

[33] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 305–319.

[34] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2016. Making fast consensus generally faster. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 156–167.

[35] Denis Rystsov. 2018. CASPaxos: Replicated State Machines without logs. *arXiv preprint arXiv:1802.07000* (2018).

[36] Yee Song, Robbert Van Renesse, Fred Schneider, and Danny Dolev. 2008. The building blocks of consensus. *Distributed Computing and Networking* (2008), 54–72.

[37] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42.

[38] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42.

[39] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 68–81.

[40] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: high-availability storage made practical in WeChat. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1730–1741.