ABSTRACT
The optimal read strategy for strong consistent key-value applications is to enable the per-replica local reads that each replica has the ability to serve reads locally. Unfortunately, current schemes for the per-replica local reads are perplexed by two issues. First, some schemes have to violate the per-replica local reads when the workload is skewed, degrading the throughput. Second, most of current schemes reply on leases or a specialized hardware to guarantee the linearizability, bringing difficulties to the deployment.

In this paper, we propose Glean, a linearizable read protocol that solves the issues of current schemes. In Glean, replica nodes always serve reads locally and we ask clients to validate the linearizability. To achieve the validation, Glean designs a novel read algorithm that allows the client to glean a consensus hint from replicas and enables replicas to contribute to the validation lightweight and fast. We implement Glean with a widely-used software stack. Our 3-replica evaluation shows that the throughput of Glean is at most 2.1× to the throughput of an unreplicated application under heavy-read workloads.

1 INTRODUCTION
Strong consistent key-value applications (SCKVAs) play an important role in network, distributed storage and generic coordination services, constructing the foundation of reliable cloud systems. One SCKVA consists of a group of replicas and utilizes a consensus protocol to keep replicas consistent and provides a high available service. Most of industry SCKVAs [1, 4, 6, 16, 17] adopt single-leader consensus protocols (like Multi-Paxos [19] or Raft [26]) for the simplicity and deployability. Unfortunately, it’s well known that vanilla single-leader consensus protocols have weakness on the throughput, especially for read operations, which consequently hurts the performance of the whole infrastructure.

To overcome this drawback and meet the needs of real read-mostly scenarios, many applications implement a leader lease protocol [4, 6, 16, 17] to improve the throughput of reads. However, the leader lease only allows one replica to process reads locally. In order to fully utilize the capability of all replicas, it is imperative to enable the per-replica local reads, which is known to be a challenging task under the requirements of linearizability [15].

Two recent schemes achieve the per-replica local reads under the context of consensus protocols: follower lease [7, 23] and switch-based reading [18, 30]. However, both schemes face the issue of throughput degradation under conflicts. Both schemes extend the single-leader consensus protocol and allow all replicas handle reads. Since the read request and the write request are served by different replicas, there exists a point that the read needs to wait the completion of the write when they target to the same key. Thus, the throughput drops until conflicting writes are finished.

Besides, current schemes face the problem of deployment. For the first scheme, the uncertainty of clock skew, attacks and misconfigurations on local timer all endanger the linearizability and complicate the implementation [4, 11], and the latter two cases are irreparable even machines are equipped with the latest clock synchronization techniques like Graham [24]. For the second scheme, the programmable switch is only available in advanced data centers. Thus, the second scheme is not fitted in WAN and under-upgrade data centers. The second scheme also brings issues for network topology design and congestion avoidance due to the centralized role of the programmable switch.

Epaxos [22] is a multi-leader consensus protocol and it also can be installed leases to enlarge the capacity of serving reads. However, the detail of how to manage leases with EPaxos is unspecified and it still faces the problem of deployment.

In this paper, we propose Glean, a linearizable read protocol for SCKVAs, which extends the leader-based consensus and acts like a multi-leader protocol. Glean enables the per-replica local reads while it is free of leases and centralized in-network hardware. The core of Glean consists of three pieces:

(1) A data structure called guidance that splits the leadership of single-leader consensus, guides read and write requests and helps to validate the linearizability.

(2) A client-side algorithm that validates linearizable reads. The algorithm requires the client to glean the guidance from the majority of replicas to check the leadership. When there is no failure and the leadership is stable, the client can always finish a read in 1 RTT.

(3) The parallel nature of reading values and reading the guidance on the replica. Reading the guidance is lightweight since and occupies little computation resources.

In this paper, the terminology “replica” refers to the replicated data and the server process managing that replicated data.
its size is small. Thus, each replica performs close to an unreplicated application on serving reads.

We implement Glean with a commonly-used software stack on commodity machines. Our evaluation under the 3-replica setting shows that Glean can achieve up to 2.1× higher throughput than an unreplicated application, and 7× higher throughput than Raft. Glean performs better than the unreplicated as long as the read ratio is higher than 75%. Glean also keeps the throughput steady when the workload contains random hotspots, which represents a skewness on the data requests. Further experiments show that the throughput degradation under one random failure is moderate and the throughput can recover to the original level within a short period if replicas are not overloaded.

2 BACKGROUND

SCKVAs need to be consistent and highly available. Consistency has twofold meanings. First, it requires the data of each replica to be consistent. Second, the SCKVAs need to be linearizable. High availability means the application needs to keep working and replying clients even if some parts of the system fails [2, 13, 14, 28].

2.1 Consensus Protocols

SCKVAs [1, 4, 17] can be easily implemented with consensus protocols since the safety and liveness of consensus protocols are highly related to the consistency and high availability of SCKVAs. Consensus protocols can be classified into two categories: the single-leader protocol [19, 25, 26] and the multi-leader protocol [19, 21, 22].

In single-leader protocol, a stable leader is in charge of building an identical operation sequence for all replicas in the state machine replication (SMR) log. Besides the leader, other replicas are called followers. The leader replicates all operations it receives to followers by a safety rule. The protocol guarantees there is only one exclusive leader serving clients by using a logical timer, which is a term in Raft [26]. In multi-leader protocol, each replica can take over a portion of the responsibility of the stable leader and the SMR log does not have to contain only one sequence [22].

2.2 Linearizability

Linearizability [15] forces a replicated key-value application to behave like one unreplicated key-value application. It means each operation takes effect instantaneously and exactly once at some point in time between its invocation and completion. Once a read operation has seen a value then all subsequent operations must see the same value or a new value that is written by another write operation. If one read is concurrent with one write and the invocation of the read advances the completion of the write, then the read is allowed to get the previous value before the write since the read may take effect before the write. Checking the operation history is the essential methodology of validating the linearizability. Figure 1 demonstrates what case satisfies the linearizability and what case violates the linearizability.

2.3 Current Linearizable Read Schemes

Traditionally, the leader is installed with a leader lease [4, 6, 26] to process reads locally, making it to perform as an unreplicated application. Since the leader always retains the latest values within the lease time, it can reply correct value to the client. Follower lease [7, 23] is proposed for replacing the leader lease. The leader can install leases on followers for stable keys and revoke leases form followers when a write request wants to update the key. The follower with the lease can serve reads locally. EPaxos [22] also can install leases to replicas for the per-replica local reads to make the per-object leader become stable.

CURP [27] and PQR [8] abandon the idea of the lease and achieve similar throughput as the leader lease. Nevertheless, they do not enable the per-replica local reads so the throughput is less than the Follower lease and EPaxos with lease.

FLAIR [30] uses a programmable switch to conduct the read request to one proper follower, then the follower process reads locally and replies. However, it still faces the problem that the follower can not serve reads during a conflicting write. CURP with read optimization [27] and Hermes enable the per-replica local reads without using leases or specialized hardware. But they fall out of the scope of consensus protocols, and they also have to stall the read on serving concurrent and conflicting writes.
3 GLEAN PROTOCOL

We propose Glean, a high-throughput linearizable read protocol for SCKVAs which extends leader-based consensus protocols and acts as a multi-leader protocol. It is co-designed with SCKVAs and always serves reads locally on one replica. Glean guarantees liveness if the majority of replicas are non-faulty [2, 13]. We consider single-key read and write operations, including GET, SET, DELETE and read-modify-write (RMW) operations, where GET is the read operation and others are write operations. We proof the correctness of Glean by TLA+.

![Figure 2: Basic idea of Glean.](image)

3.1 Basic Idea

The basic idea of Glean is fairly straightforward as shown in Figure 2. This basic idea works together with the single-leader consensus protocol, like Raft, that should establish a correct leader. We introduce the basic idea with only one key-value pair.

At the first phase, the client sends the read request to the leader as well as \( N - 1 \) check-term requests to followers at the same time. The leader processes the request locally and returns the value and the term to the client. The follower returns the term. At the second phase, the client gleans results from the leader and followers. If \( \lceil \frac{N-1}{2} \rceil \) followers have the same term as the term of the leader, then the result of the leader is validated and the client finishes the read. However, as long as one follower returns a term that is bigger than the term of the leader before the read quorum is reached, the client should continue the read from the first step until the read is finished. On processing the check-term request, the follower does not need to access its SMR log or storage. Hence, the check-term request can be processed fast and in parallel with other requests.

Our basic idea differs from PQR [8] in that we do not check each consensus entry and read from followers, instead we check the correct leadership of the leader and read from the leader.

**Correctness.** The correctness is obvious when the leadership is stable, so our basic idea only jeopardizes the linearizability when the leader is changing and at the same time one read is initiated. We prove the linearizability of our basic idea by TLA+. Here, we provide an intuitive explanation. Assume \( \lceil \frac{N-1}{2} \rceil \) followers and the leader form a quorum \( Q \). Replicas in \( Q \) return the same term \( γ \) within a time period \( T \), while a successful leader election with of term \( γ + 1 \) intersects with \( T \). Then, at least one replica in \( Q \) return term \( γ \) before voting to the new leader by advancing its local term to \( γ + 1 \). Thus, the new leader must be elected after the start point of \( T \), which is also after the invocation point of the read request. So any write that changes the value from the new leader must be sequential behind or concurrent with the read. According to the definition [15], the operation history of reads and writes satisfies the linearizability.

3.2 Glean’s Guidance

The **guidance** is the core data structure of Glean, which makes our basic idea to transform into a multi-leader protocol. It consists of two parts: the term and a mapping of keys to replicas. The term is the same meaning as it is in the leader-based consensus protocol, which is used for safe replication and checking leadership. The guidance is comparable by comparing the term. The mapping is represented as partitions on the key space. We do not restrict the form of partitioning. Nonetheless, each partition should be exclusive and all partitions must constitute the whole key space. The generation of guidance should ensure that one term of guidance only has one mapping rules, which can be achieved by using another consensus instance. Each replica has a copy of guidance.

**Per-Partition leader (P-leader).** Each partition is leaded by one replica, which is called P-leader. All requests that target to keys in one partition should be sent to one P-leader. The P-leader is in charge of replicating write requests to other replicas. On receiving a replication message from one P-leader, the replica should behave like a follower in leader-based consensus protocols. Unlike partitioning strategies that aim for scalability in multiple replica groups [3, 5, 16, 20, 29], our goal of partitioning is the per-replica local reads inside one replica group.

The introduction of P-leader leads to partitioning on the SMR log. We assign multiple key groups where each key group is a composition of multiple keys, and each P-leader leads multiple key groups. In our implementation and evaluation, we find that 16 key groups is enough for good partitioning and load balance under the 3-replica setting. Unlike multi-leader protocols like Mencius [21] that splits the log by the index number, we split the log by key groups. Unlike EPaxos [22] that each replica has a corresponding sub-log, we grant each key group a sub-log. Although our strategy
3.3 Client-Side Algorithm

On starting, each client collects the same guidance from \( \lceil \frac{N+1}{2} \rceil \) replicas and caches the guidance. The concept of P-leader can be used for deciding the destination of client requests. Since the guidance contains the mapping of keys to P-leaders, the client can choose the P-leader when sending a request of one key. An example of how clients initiate requests is shown in Figure 3. Each response contains the guidance of that replica.

**Write operation.** The client sends one write request to the P-leader. The write path involves the replication from one P-leader, which requires the same safety rule of the single-leader consensus protocol. If the replica that receives the write request finds itself is not the P-leader of the requested key, the replica should reject the request and return the new guidance to the client. This situation only happens when the client caches a stale guidance.

**Read operation.** The read strategy is similar to our basic idea in Section 3.1, while the client needs to collect the same guidance among \( \lceil \frac{N+1}{2} \rceil \) replicas including the P-leader. The message for collecting the guidance is called GetGuidance. A read ID can also be appended into each read request and GetGuidance messages to avoid the client receiving stale responses.

The client uses two phases to finish a read. The first phase is the send phase that the client sends 1 read request to P-leader and \( N - 1 \) GetGuidance messages to other replicas. The second phase is the glean phase, which contains three steps.

1. **Step 1.** The client waits for responses from any of replicas. If the response is from the P-leader, the client records the value and the guidance. If the response is not from the P-leader, the client collects the guidance into a set called the waiting set.
2. **Step 2.** The client checks if the guidance of the P-leader or any guidance in the waiting set is larger than its local cached guidance. If such a guidance does exist, the client stop current phase and marks the glean phase as failed.
3. **Step 3.** If there are \( \lceil \frac{N-1}{2} \rceil \) guidances in the waiting set that are the same as the guidance of the P-leader, then the client validates the value and quits the glean phase with s status of success. Otherwise, the client keeps executing the step 1.

If the glean phase fails due to a new guidance is found, the client retries the send phase with the new guidance. The client should set a timeout timer and retry the read when the timer is triggered. The client keeps at most one outstanding operation at one time. In real implementations, all above algorithms should be written into a client library.

3.4 Replica-Side Behaviours

Each read request, write request and inter- replica message needs to be handled sequentially. Although the strategy of partitioning hints that messages target to a key can be processed in a separate thread, the amount of threads cannot fit the amount of keys. On a multi-core server, tens or hundreds of CPU cores are the limit, while the amount of keys may be up to millions [12]. Thus, messages that target to some keys have to be processed on one core. To avoid the overhead of scheduling threads, one thread has to process messages target to multiple keys and processes them sequentially.

The crux of high throughput is that GetGuidance can be handled in parallel with other messages because the handling procedure does not involve the SMR log or the storage. Thus, the processing capacity of one replica is fully occupied by local reads under 100% read case, which is exactly the same as an unreplicated application. On each replica, the guidance is stored in a fixed address in memory, like a global variable. Thus, the retrieving of the guidance may be completed by one-sided RDMA.

**Glean++.** We introduce an optimized Glean called Glean+ to reduce the amount of GetGuidance. When there are multiple keys waiting to be read and each of them targets to a distinct P-leader, the client initiates multiple read requests concurrently to the corresponding P-leaders. The client waits all requests returned to validate reads. This strategy is not batching since each replica still execute one operation for one read request. This optimization can work for at most \( N \) consecutive reads.
4 IMPLEMENTATION

We implement the prototype of Glean using the Go programming language. We use Linux kernel-based TCP stack since it is a widely-used module for CSDKA [1, 16, 17]. The storage is in-memory. Each replica has one main thread to handle read and write requests, as well as inter-replica messages. Receiving and sending are asynchronous to the main thread so the main thread only processes protocol algorithm. We set the number of key groups in the guidance to 16 and the replication messages of each key group are transmitted to another replica by a dedicated TCP connection. Thus, there are 16 TCP connections between any two replicas. The reason we implement multiple TCP connections is that we find the average time of calling write(socketfd) function is 5 μs, which is close to the average service time of replication and it may block the main thread for a few microseconds.

To handle GetGuidance, each replica has a special thread besides the main thread, which we call the guidance thread. The guidance thread only retrieves local guidance to fill the response of GetGuidance. We use a memory barrier to protect the guidance. Since the changing of the guidance is rare, the access to the guidance is as quick as the cache speed.

5 EVALUATION

We conduct comprehensive experiments for Glean using the implementations introduced in Section 4. We also implement Raft and an unreplicated in-memory key-value store with identical software settings as Glean. All implementation have only one main thread for serving read and write requests.

Testbed. Our real-world testbed includes 6 commodity machines. All of them are based on Intel Xeon Gold Processor, 256GB DDR4 memory and 10 Gbps interconnects. The installed operating system is Ubuntu 20.04 LTS. Three of them are used for replicas and the other three are used for clients. They are connected by one LAN and the average RTT of two machines is 110 μs tested by ping.

Workload. Each client machine runs multiple virtual clients to generate workloads, and reports statistics periodically to one statistic server. Each virtual client keeps at most one outstanding operation at each time point. The workloads are similar to YCSB [10] but have more fine-grained control to achieve more precise evaluations. We pre-install 1 million key-value pairs and each of them is of 1K bytes size on one replica. Unless otherwise specified, the workloads randomly select 4096 keys among all installed keys and visit them repeatedly. The throughput and latency are measured on clients. The parameter θ controls the skewness, and θ = 0 means the uniform distribution.

5.1 Throughput

We evaluate Glean+, Glean, Raft [26] and the unreplicated. The target keys are randomly generated from the key space, except for the controlled Glean+ case. We steadily increase the number of virtual clients, until the throughput stops rising and the latency starts rising.

Read-only workload. The result under 100% read workload is shown in Figure 4. The controlled Glean+ is achieved by controlling each virtual client to generate a workload, making any three consecutive reads are destined to different P-leaders that fits the ideal scenario of Glean+. The controlled Glean+ can achieve 660 Kops/s throughput, which is exactly 3× to the unreplicated. Glean+ achieves 520 Kops/s peak throughput and Glean achieves 460 Kops/s, that is 2.36× and 2.09× to the unreplicated. Vanilla Raft performs poorly since the leader replicates each operation to followers, which makes the throughput of Raft is only 30% to that of the unreplicated. The result is steady even we change the key size (Figure 5).

Varied read ratios. We evaluate how Glean performs under varied read ratios, as shown in Figure 6. Glean provides higher throughput than the unreplicated when the read ratio is higher than 75%. The throughput of Glean drops quickly as the read ratio decreases. Glean only has 60% throughput to that of the unreplicated under the 50% read workload. Since Glean relies on leader-based SMR to finish write operations,
the dropping of throughput is in expectation. However, Glean still outperforms Raft by 95% under the 50% read workload.

The throughput of the unreplicated is stable in spite of the read ratio changes. The same situation happens for Raft. That is because we use in-memory storage, and the structure of the key-value pair is succinct. Hence, the read and write for a single key almost take the same time.

**Varied skewness.** Next, we evaluate how Glean performs under hotspots under the 95% read workload (YCSB heavy read). Data hotspots may increase the possibility of interference, since more concurrent operations to one key arise. The zipfian parameter $\theta$ changes from 0.5 to 1.2 since recent operational experience from Alibaba [9] hints that widely-believed extreme cases like $\theta = 1.2$ should also be considered. The result (Figure 7) shows that Glean provides stable throughput across all skewness cases, and the throughput gets even higher with increasing skewness. This result can be explained by two reasons. First, since we generate workloads randomly, the popular keys are distributed almost evenly in the whole key space so all P-leader of Glean experience a portion of hot keys. Second, as the skewness becomes more significant, the hotspots in CPU cache make the reads and writes to be processed faster.

### 5.2 Service Time

We evaluate the service time of different schemes and the result is shown in Figure 8. The number after the “-” line represents the read ratio of Glean. For Raft and the unreplicated, we omit their performances under multiple read cases since they have stable performance under all cases. In our evaluation, Glean has a longer service time tail than that of the unreplicated. This tail is caused by the overhead of software stack because the Glean application receives more network packets, the GetGuidance messages, than the unreplicated. The average service time of the guidance thread (1.4 $\mu$s) is three times lower than that of the main thread in Glean-100 (5.2 $\mu$s). Since one read request is related to two GetGuidance messages in the 3-replica setting, one guidance thread is fast enough to handle messages.

For Raft, the leader needs to handle client messages and replica messages in the same event loop, which represents a two-step characteristic for SMR. The follower takes a shorter time than the leader because it has lower amount of loading and more concise logic. This two-step characteristic also applies to Glean and Glean+, causing a longer processing time under heavy-write workloads.

### 5.3 Availability and Fault Tolerance

We evaluate how Glean performs under a failure. Clients are grouped into three groups and each group only initiates reads to one P-leader. We randomly shut down one replica and observe how the throughput changes. We conduct two evaluations with two different loading settings: mild and heavy. As shown in Figure 9, when we shut down one replica, the throughput drops about 33%. A short period after the crash, the timeout of the crashed replica is detected and the guidance is changed by a replicated state machine service.

![Figure 9: Throughput changes after a replica crash.](image)

**Mild loading.** We start 18 virtual clients. Under mild loading, none of the replicas are overloaded, and the latency is kept at 170 $\mu$s in the beginning. After the detection of the crash, the throughput and the latency recover to the previous level, because none of the replica is overloaded.

**Heavy loading.** We start 225 virtual clients to make replicas almost overloaded. The latency is kept at 430 $\mu$s in the beginning. After the crash detection, the throughput changes to about 66% of the original level, and the latency rises to 550 $\mu$s. Unlike in the mild case, the throughput does not restore to the original level because now the replicas are overloaded.

### 6 CONCLUSION

We propose Glean, an extension to the traditional leader-based consensus protocol. Glean solves the problem of high-throughput linearizable read of SCKVAs with a novel read mechanism both on the client side and the replica side without compromising on usability. We implement the prototype of Glean with commonly used software modules to show its benefits.
REFERENCES


