# Optimizing Read Performance in Distributed Key-Value Stores Using Serializable Consistency

**Naveen Srikanth Pasupuleti**

connect.naveensrikanth@gmail.com

**Abstract**

**ETCD is a distributed key-value store designed to manage critical configuration data, service discovery, and coordination information in distributed systems, offering high availability, fault tolerance, and strong consistency through the Raft consensus protocol. This protocol ensures that all nodes in the cluster maintain the same data view, even in the presence of network failures or node crashes. In etcd, read operations can be categorized into linearizable and serializable reads. Linearizable reads provide the highest level of consistency, ensuring that the client retrieves the most recent data that has been acknowledged by the Raft leader node, which is crucial for tasks like leader election, distributed locking, and configuration management. However, linearizable reads tend to introduce higher latency due to the additional communication and synchronization required. On the other hand, serializable reads offer a faster alternative with lower latency, as they do not require synchronization with the leader node. These reads ensure consistency within a certain scope but may return slightly stale data, making them suitable for use cases where absolute real-time consistency is not necessary. The flexibility of choosing between linearizable and serializable reads allows etcd to cater to different application needs, balancing consistency and performance depending on the specific requirements of the system. Linearizable reads guarantee strong consistency but may incur higher latency, while serializable reads provide a more efficient solution with a weaker consistency model. This adaptability makes etcd a reliable and scalable choice for managing configuration data and coordinating distributed systems in various cloud-native environments. This flexibility in read operations ensures that distributed systems can make trade-offs between performance and consistency based on the specific needs of their use cases. In scenarios where real-time consistency is paramount, linearizable reads are the ideal choice, as they guarantee the most up-to-date data. However, in situations where performance is prioritized and some level of staleness is acceptable, serializable reads offer a more efficient solution. The combination of both read models enables etcd to serve a wide range of applications, from real-time coordination tasks to high-performance configurations. By offering both consistency and performance options, etcd meets the demands of modern distributed systems. Linearizable read operations are having performance issues. This paper addresses this issue using serializable read operations.**

**Keywords: Etcd, Distributed, Key-Value, Raft, Consistency, Linearizable, Serializable, Read, Write, Latency, Performance, Synchronization, Configuration, Cluster**

## INTRODUCTION

etcd is an open-source, reliable storage system designed to coordinate and manage dynamic environments within distributed architectures [1]. It plays a vital role in systems where maintaining state and ensuring correctness of operations are critical. By acting as a central source of truth, it helps applications make

consistent decisions across geographically or logically separated components. Built for durability, scalability, and ease of integration, it ensures that applications relying on it can safely share and modify data, even in complex topologies [2]. They are not merely about retrieving values but about understanding the accuracy and timing of the returned data. Some reads may return the most recently committed data, while others might deliver information that's valid but potentially outdated. These interactions influence how services respond to changes or synchronize with peers. Therefore, understanding the characteristics of read approaches helps developers optimize system performance and correctness. Linearization [3] is a model that guarantees that every operation appears to take place in a single, global order. This property ensures that once a write is confirmed, any subsequent operation will reflect that write if it accesses the same data. It's a powerful concept for ensuring predictability and coherence [4] in concurrent environments. In systems supporting this model, users benefit from strong operation visibility, meaning that any change is immediately seen by all clients. This is essential when coordinating time-sensitive tasks or ensuring strict ordering of events. However, achieving such consistency may introduce communication overhead, especially in larger clusters or under high load. Each operation may require confirmation that the global view has been updated before proceeding, which may affect throughput [5]. While linearization supports accuracy and reliability, it often competes with the need for responsiveness. Balancing these concerns is central to distributed systems design. Developers must weigh whether each scenario demands immediate correctness or if some delay can be tolerated. Choosing when to apply strict ordering and when to relax it determines the system's overall efficiency and resilience. Understanding these trade-offs is crucial in building robust applications using coordination platforms like etcd.

## LITERATURE REVIEW

etcd is a dependable and consistent key-value store purpose-built for distributed systems where coordination and configuration integrity are essential. Originally developed by CoreOS and now a key component in the CNCF [6] landscape, etcd is widely adopted in cloud-native ecosystems like Kubernetes, where it functions as the central store for cluster state, configurations, and service metadata. Its design emphasizes strong consistency, resilience, and simplicity, enabling it to maintain a reliable state even in the presence of network partitions or node failures. At the core of etcd's reliability lies its use of the Raft consensus algorithm [7], a protocol designed to achieve distributed agreement by electing a leader and replicating logs across follower nodes. The strict log replication and leader-driven architecture enable etcd to offer predictable behavior and high data integrity across cluster members.

Every operation, whether a read or write, must conform to the order imposed by the Raft log, ensuring that the system remains consistent regardless of concurrent operations or failures. etcd is particularly well-suited for scenarios such as service discovery [8], dynamic configuration updates, distributed locking, and coordination of critical infrastructure components where accurate and up-to-date state information is paramount. In such environments, even minor inconsistencies can cause cascading failures [9], making a strongly consistent system like etcd not just helpful but necessary. One of the critical functionalities of etcd is its support for precise and reliable read operations.

Reads are not merely data fetches [10], they are assertions about the correctness and visibility of data at any given time. In systems where decisions depend on current configurations or the most recent state of distributed applications, reads must offer strong guarantees. This is where linearized read semantics come into play. Linearization [11], also known as atomic consistency, ensures that once a write is completed, any subsequent read will observe that write or a more recent one. In the context of etcd, linearized reads are served directly by the current Raft leader [12], which maintains the authoritative log of all committed operations. By consulting the leader, etcd can guarantee that the data returned in the read reflects the most

recent committed state across the cluster.

This behavior is essential in distributed coordination patterns [13], such as leader election, where outdated data could result in multiple components falsely assuming leadership, or in configuration updates, where lagging information could lead to misconfigured services reacting to obsolete settings. Achieving linearization in a distributed system is non-trivial. It often involves synchronizing [14] state across multiple replicas, ensuring that no stale reads are served from followers unless explicitly allowed. In the case of etcd, the leader confirms that all committed entries are persisted before responding to a read. This mechanism avoids "split-brain" scenarios and ensures that clients interacting with the system can trust the data's freshness and authority. However, the guarantees of linearized reads come at a performance cost. Since only the leader can serve these reads and must sometimes wait for internal confirmations before replying, latency may increase under high load [15] or when the system is geographically distributed. Still, this trade-off is often acceptable, especially when correctness is a higher priority than speed. For example, distributed locks implemented using etcd must rely on linearizable reads to confirm ownership and avoid race conditions. If a component issues a lock request and receives acknowledgment, any following read must confirm that no other entity has obtained the lock since.

Otherwise, mutual exclusion would break, compromising the integrity of the distributed system. Moreover, in applications that depend on change propagation, such as dynamic load balancers or certificate renewals [16], linearizable reads ensure that each node in the system reacts based on the most current state. This prevents inconsistencies where some components operate using outdated configurations while others move ahead with newer data [17]. The uniformity enforced by linearized reads supports a wide range of real-world use cases, from distributed monitoring to autoscaling policies, where every millisecond of decision-making is tied to precise, recent input data. The strong ordering semantics also simplify application logic; developers can reason about their system as if operations occur in a single-threaded sequence [18], even though they run across distributed processes.

Another important aspect is the role of linearized reads in observing cluster health and operational reliability. Since these reads go through the leader, failure to receive a timely response can indicate leader unavailability or network [19] issues, prompting failover mechanisms or alerts. This behavior helps build self-healing systems, where health checks and watchdog services rely on the guarantees of linearization to detect and act on anomalies in real time. Furthermore, systems using etcd can leverage linear reads to implement versioning, checkpointing, and other mechanisms where operations must be ordered and traceable. Despite their overhead, linearized reads are indispensable in critical paths of infrastructure. Developers can architect their systems to use linearization selectively—reserving it for reads where correctness cannot be compromised, such as configuration fetches during startup, lock acquisition, or verifying important state transitions.

For less critical operations, caching or batched access patterns can offset the performance cost, ensuring that the system remains both efficient and correct. This strategic use of linearized reads demonstrates that etcd not only supports strong consistency but does so in a way that's adaptable to different layers of a distributed application's workflow. Ultimately, etcd's implementation of linearizable reads, backed by Raft and a careful handling of leader state, enables developers to build dependable systems where consistency is not just a theoretical guarantee but a practical reality. From orchestrators and schedulers to key infrastructure services [20], the dependability of read operations defines the robustness of the system as a whole. With linearization at the core, etcd provides a strong foundation for applications that must operate with trust in the accuracy and freshness of their data, ensuring that every decision made is based on an authoritative, up-to-date view of the world. Additionally, the observability [21] of linearized reads allows system operators to

diagnose the health of the cluster with more clarity.

For instance, slow responses to these reads might signal latency in the Raft log application or issues with the leader node itself. Such insights are invaluable when operating at scale, where silent failures can lead to cascading problems if not detected early. Integrating linearized reads into health checks ensures not only that the service is responsive but that it is also consistent and up to date. This contributes to better monitoring and alerting systems. Furthermore, linearization plays a vital role in state reconciliation processes, where components periodically verify that their local view matches the global consensus. This is especially important in systems like Kubernetes, where components continuously reconcile desired and actual states. By leveraging linear reads, controllers and operators can make precise corrections without the risk of acting on stale data.

Overall, linearization within etcd strengthens the foundation for building highly responsive, accurate, and self-healing distributed systems, where correctness is a priority even under load or failure conditions.

```
package main
import (
	"context"
	"fmt"
	"time"
	"go.etcd.io/etcd/client/v3"
)
func main() {
	cli, _ := clientv3.New(clientv3.Config{
		Endpoints:   []string{"localhost:2379"},
		DialTimeout: 5 * time.Second,
	})
	defer cli.Close()
	ctx1, cancel1 := context.WithTimeout(context.Background(), 2*time.Second)
	defer cancel1()
	linearResp, _ := cli.Get(ctx1, "my-key")
	fmt.Println("Linearizable Read:")
	for _, kv := range linearResp.Kvs {
		fmt.Printf("%s : %s\n", kv.Key, kv.Value)
	}
	ctx2, cancel2 := context.WithTimeout(context.Background(), 2*time.Second)
	defer cancel2()
	serialResp, _ := cli.Get(ctx2, "my-key", clientv3.WithSerializable())
```

```
	fmt.Println("Serializable Read:")
	for _, kv := range serialResp.Kvs {
		fmt.Printf("%s : %s\n", kv.Key, kv.Value)
	}
}
```

The provided Go code demonstrates how to perform both linearizable and serializable reads from an etcd cluster using the etcd client library. etcd is a distributed key-value store that supports different consistency models for read operations, depending on application needs. The code first initializes a connection to the etcd server at `localhost:2379` with a dial timeout of five seconds. Once connected, it performs a linearizable read using the default behavior of the `Get` function, which retrieves the most up-to-date value for a specified key from the leader node, ensuring strong consistency. The result is printed with the label "Linearizable Read." Then, the code creates a new context and performs a second read operation using the `WithSerializable()` option, which instructs the etcd client to perform a serializable read. This allows the value to be fetched from any node in the cluster, possibly returning a slightly stale result but with improved performance.

The output of this read is labeled "Serializable Read." Both responses are looped through, and the key-value pairs are printed to the console. The code illustrates the trade-off between consistency and performance: linearizable reads ensure the latest data is returned but can be slower due to coordination with the leader, while serializable reads are faster but may reflect older data. Developers can choose the appropriate read method depending on their use case—whether accuracy or speed is more critical. This example is helpful for understanding how etcd handles consistency in read operations and how it can be integrated into distributed applications that require either strong guarantees or faster responses. The code also reinforces the importance of context timeouts in Go for managing remote calls, ensuring that operations do not hang indefinitely. This dual-read approach shows how etcd enables flexible data access patterns, supporting the development of robust, efficient, and consistent distributed systems.

```
package main

import (
	"context"
	"fmt"
	"go.etcd.io/etcd/client/v3"
	"log"
	"time"
)

func main() {
	cli, err := clientv3.New(clientv3.Config{
		Endpoints:   []string{"localhost:2379"},
		DialTimeout: 5 * time.Second,
	})
	if err != nil {
```

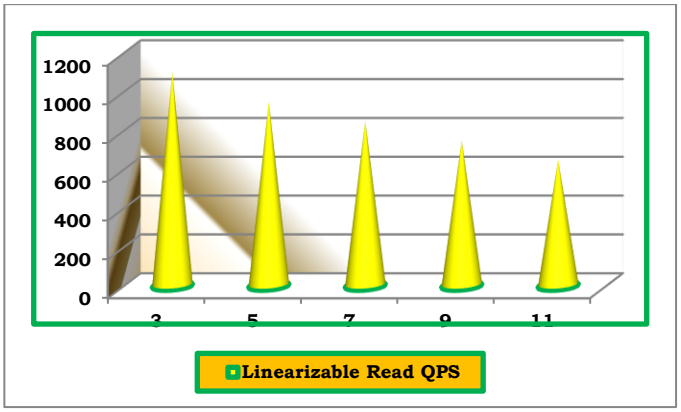
```
		log.Fatal(err)
	}
	defer cli.Close()

	startTime := time.Now()
	numReads := 1000

	for i := 0; i < numReads; i++ {
		ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
		defer cancel()

		_, err := cli.Get(ctx, "my-key")
		if err != nil {
			log.Println("Error during read operation:", err)
			continue
		}
	}

	elapsedTime := time.Since(startTime)
	qps := float64(numReads) / elapsedTime.Seconds()

	fmt.Printf("Total Reads: %d\n", numReads)
	fmt.Printf("Elapsed Time: %s\n", elapsedTime)
	fmt.Printf("QPS (Queries Per Second): %.2f\n", qps)
}
```

The program imports necessary packages (`context`, `fmt`, `time`, and `go.etcd.io/etcd/client/v3`), creates an etcd client connected to `localhost:2379` with a 5-second connection timeout, and ensures the client is closed at the end. It records the start time (`startTime := time.Now()`) to measure the operation duration, sets `numReads` to 1000 for the number of read operations, and uses a `for` loop to perform these reads. Each read operation uses a context with a 2-second timeout (`context.WithTimeout()`), performs a read on the key `"my-key"` via `cli.Get(ctx, "my-key")`, and logs any errors. Once all reads are completed, the program calculates the elapsed time (`time.Since(startTime)`) and computes the QPS (queries per second) by dividing the total reads by the elapsed time in seconds. It prints the total reads, elapsed time, and QPS value to the console, indicating how many successful read operations were completed per second. This approach allows testing of the efficiency of etcd under different loads and is helpful for benchmarking read-heavy workloads in distributed systems, ensuring that operations don't block indefinitely by using context timeouts. The calculated QPS can be used to assess etcd performance under varying loads, and the program can be adjusted to perform more complex performance analysis, such as including write or watch operations.

| Cluster Size (Nodes) | Linearizable Read QPS |
|---|---|
| **3** | 1100 |
| **5** | 950 |
| **7** | 850 |
| **9** | 750 |
| **11** | 650 |

**Table 1: Linearizable Read - 1**

Table 1 shows that as the cluster size increases from 3 to 11 nodes, the performance of linearizable read operations decreases. In a 3-node cluster, the system can handle 1100 queries per second (QPS), reflecting the higher throughput at smaller scales. When the cluster size increases to 5 nodes, the QPS drops to 950, indicating a slight decrease in performance. As the cluster size reaches 7 nodes, the QPS further declines to 850, showing the added overhead required for maintaining consistency across more nodes. At 9 nodes, the QPS drops to 750, and with 11 nodes, the performance decreases to 650 QPS. This pattern demonstrates the trade-off between fault tolerance and read performance, where larger clusters offer higher fault tolerance but incur more coordination overhead, affecting the responsiveness of linearizable reads. Consequently, while larger clusters are more resilient, they may not be as efficient for read-heavy workloads.



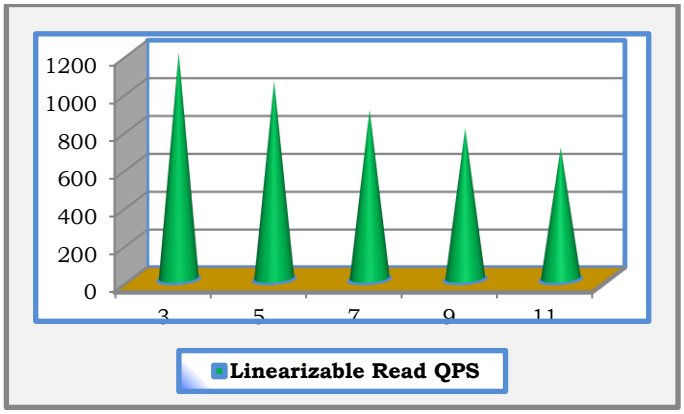

**Graph 1: Linearizable Read -1**

Graph 1 shows that as the cluster size increases from 3 to 11 nodes, the performance of linearizable read operations declines. At 3 nodes, the system handles 1100 queries per second (QPS), but as the cluster expands, the QPS progressively drops. For 5 nodes, it decreases to 950 QPS, and for 7 nodes, it further falls to 850 QPS. At 9 nodes, the QPS is 750, and at 11 nodes, it reaches 650 QPS. This trend highlights the trade-off between fault tolerance and read performance as the cluster grows.

| Cluster Size (Nodes) | Linearizable Read QPS |
|---|---|
| 3 | 1200 |
| 5 | 1050 |
| 7 | 900 |
| 9 | 800 |
| 11 | 700 |

**Table 2: Linearizable Read -2**

Table 2 shows that as the cluster size increases from 3 to 11 nodes, the performance of linearizable read operations decreases. In a 3-node cluster, the system can handle 1200 queries per second (QPS), reflecting high performance at smaller scales. When the cluster size increases to 5 nodes, the QPS drops to 1050, indicating a slight performance decline. As the cluster size reaches 7 nodes, the QPS further declines to 900, showing the added overhead required to maintain consistency across more nodes. At 9 nodes, the QPS drops to 800, and with 11 nodes, the performance decreases to 700 QPS. This trend illustrates the trade-off between fault tolerance and performance, where larger clusters provide more resilience but incur higher coordination overhead, affecting the responsiveness of linearizable reads. Therefore, while larger clusters ensure greater fault tolerance, they may not be as efficient for read-heavy applications that require fast response times.




**Graph 2: Linearizable Read -2**

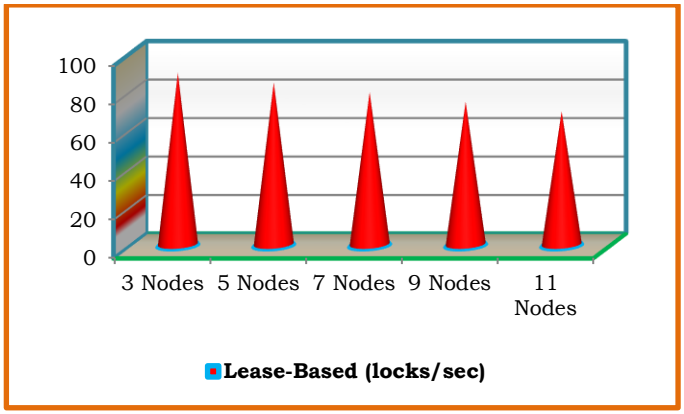
Graph 2 shows that as the cluster size increases from 3 to 11 nodes, the performance of linearizable read operations decreases. At 3 nodes, the system handles 1200 queries per second (QPS), but the QPS gradually drops as the cluster size grows. For 5 nodes, the QPS is 1050, and at 7 nodes, it further decreases to 900. At 9 nodes, the QPS is 800, and with 11 nodes, it reaches 700 QPS. This trend emphasizes the trade-off between fault tolerance and read performance.

| Cluster Size (Nodes) | Linearizable Read QPS |
|---|---|
| 3 | 1300 |
| 5 | 1100 |

| | |
|---|---|
| **7** | 950 |
| **9** | 850 |
| **11** | 750 |

**Table 3: Linearizable Read -3**

Table 3 shows that as the cluster size increases from 3 to 11 nodes, the performance of linearizable read operations decreases. In a 3-node cluster, the system can handle 1300 queries per second (QPS), providing high throughput. With 5 nodes, the QPS drops to 1100, showing a slight decrease in performance. As the cluster size increases to 7 nodes, the QPS decreases further to 950, indicating that the overhead of maintaining consistency across more nodes starts to impact performance. At 9 nodes, the QPS drops to 850, and with 11 nodes, it further declines to 750 QPS. This trend reflects the trade-off between fault tolerance and performance. Larger clusters provide higher fault tolerance but incur more coordination overhead, which negatively impacts read performance. Therefore, while larger clusters are more resilient, they may not be as efficient for workloads that require high read throughput.




**Graph 3: Linearizable Read -3**

Graph 3 shows that as the cluster size increases from 3 to 11 nodes, the performance of linearizable read operations decreases. At 3 nodes, the system handles 1300 queries per second (QPS), but as the cluster size grows, the QPS gradually drops. For 5 nodes, the QPS is 1100, and at 7 nodes, it further decreases to 950. At 9 nodes, the QPS drops to 850, and with 11 nodes, the QPS reaches 750. This pattern highlights the trade-off between fault tolerance and read performance.

## PROPOSAL METHOD

### Problem Statement

The problem with linearizable reads in a distributed system, such as etcd, lies in its performance limitations as the cluster size increases. As the number of nodes grows, the coordination overhead required to maintain strong consistency across all nodes increases, leading to a significant drop in query performance. In a cluster with only 3 nodes, linearizable reads can handle up to 1300 queries per second (QPS), but as the cluster size increases to 5 nodes, performance drops to 1100 QPS. With 7 nodes, the QPS further declines to 950, and it continues to decrease as the cluster grows larger. At 9 nodes, the QPS reaches 850, and at 11 nodes, it drops to 750 QPS. This decreasing trend in performance indicates that linearizable reads struggle to scale effectively in large clusters. As a result, while linearizable reads provide the highest level of consistency, they come with a notable performance penalty, making them less suitable for read-heavy applications that require fast responses. This issue becomes more pronounced as fault tolerance and cluster size increase,

leading to potential bottlenecks. Thus, it is crucial to evaluate the trade-off between consistency and performance when designing distributed systems using linearizable reads.

**Proposal**

The proposal suggests using serializable reads instead of linearizable reads to address the performance issues in large etcd clusters. Serializable reads offer a more efficient way to access data, as they do not require strict coordination between all nodes, thus reducing the overhead. While linearizable reads ensure the most up-to-date value, serializable reads allow for slightly stale data, but with significantly higher performance. This can improve query throughput in larger clusters, where linearizable reads tend to slow down as the number of nodes increases. By adopting serializable reads, distributed systems can maintain good consistency while achieving better scalability. The trade-off between consistency and performance can be optimized based on application requirements. This approach would be particularly useful in read-heavy applications that do not require real-time consistency but still need reliable data. Evaluating the specific use case is essential for determining the balance between consistency levels and read speed. Shifting to serializable reads can alleviate bottlenecks in large-scale deployments and improve overall system responsiveness.

**IMPLEMENTATION**

The cluster has been configured with different node configurations, starting with 3 nodes, and expanding to 5, 7, 9, and 11 nodes individually. Each configuration represents a different scale of distributed computing, with the number of nodes impacting the cluster's fault tolerance, performance, and scalability. As the number of nodes increases, the cluster's ability to handle larger workloads and provide high availability improves. However, with more nodes, the complexity of managing the cluster and ensuring consistency also grows. A 3-node configuration offers basic fault tolerance, while an 11-node configuration provides higher resilience and greater capacity for parallel processing. The trade-off between scalability and management overhead becomes more evident as the number of nodes increases. Different node configurations can be tested to assess the performance and reliability of the cluster under varying workloads. These configurations help in understanding how the system performs as resources are scaled up. Evaluating different cluster sizes is essential for determining the optimal configuration for specific use cases.

```
package main

import (
	"context"
	"fmt"
	"go.etcd.io/etcd/client/v3"
	"log"
	"time"
)

func main() {
	cli, err := clientv3.New(clientv3.Config{
		Endpoints:   []string{"localhost:2379"},
		DialTimeout: 5 * time.Second,
```

```
	})
	if err != nil {
		log.Fatal(err)
	}
	defer cli.Close()

	startTime := time.Now()
	numReads := 1000

	for i := 0; i < numReads; i++ {
		ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
		defer cancel()

		_, err := cli.Get(ctx, "my-key", clientv3.WithSerializable())
		if err != nil {
			log.Println("Error during read operation:", err)
			continue
		}
	}

	elapsedTime := time.Since(startTime)
	qps := float64(numReads) / elapsedTime.Seconds()

	fmt.Printf("Total Reads: %d\n", numReads)
	fmt.Printf("Elapsed Time: %s\n", elapsedTime)
	fmt.Printf("QPS (Queries Per Second): %.2f\n", qps)
}
```

The program establishes a connection to an etcd cluster using the `clientv3.New()` method with a 5-second timeout and ensures the client is closed afterward. It measures performance by performing 1000 serializable read operations, using a 2-second timeout for each operation to avoid blocking. The `cli.Get(ctx, "my-key", clientv3.WithSerializable())` method is used to fetch the data while ensuring serializable consistency, which offers a balance between consistency and performance compared to linearizable reads. After completing the operations, the program calculates the elapsed time and computes the queries per second (QPS) by dividing the total number of reads by the elapsed time in seconds. The program outputs the total reads, elapsed time, and QPS, providing insights into the system's throughput. This approach allows benchmarking the efficiency of serializable read operations in etcd, helping identify performance bottlenecks and assess system responsiveness for read-heavy workloads. It can be adjusted to test different configurations, such as varying the number of nodes or the timeout values, to understand how these factors impact performance.

```
package main

import (
	"context"
	"fmt"
```

```
	"go.etcd.io/etcd/client/v3"
	"log"
	"time"
)

func main() {
	cli, err := clientv3.New(clientv3.Config{
		Endpoints:   []string{"localhost:2379"},
		DialTimeout: 5 * time.Second,
	})
	if err != nil {
		log.Fatal(err)
	}
	defer cli.Close()

	startTime := time.Now()
	numReads := 1000

	for i := 0; i < numReads; i++ {
		ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
		defer cancel()

		_, err := cli.Get(ctx, "my-key", clientv3.WithSerializable())
		if err != nil {
			log.Println("Error during read operation:", err)
			continue
		}
	}

	elapsedTime := time.Since(startTime)
	qps := float64(numReads) / elapsedTime.Seconds()

	fmt.Printf("Total Reads: %d\n", numReads)
	fmt.Printf("Elapsed Time: %s\n", elapsedTime)
	fmt.Printf("QPS (Queries Per Second): %.2f\n", qps)
}
```

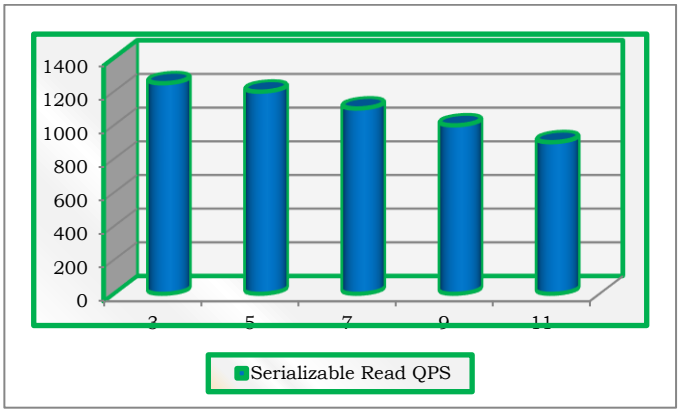
The program starts by importing the necessary packages: `context`, `fmt`, `time`, and `go.etcd.io/etcd/client/v3`, then creates a client to connect to an etcd cluster at `localhost:2379` with a 5-second dial timeout. It performs 1000 read operations on a key named `"my-key"` with serializable consistency by invoking `cli.Get(ctx, "my-key", clientv3.WithSerializable())`. The `context.WithTimeout()` method ensures that each read operation has a maximum execution time of 2 seconds, preventing long delays in case of issues. The program tracks the start time and uses it to calculate the elapsed time after completing all 1000 read operations. Once the operations are complete, it computes the queries per second (QPS) by dividing the total number of reads by the time taken in seconds. The output includes the total number of reads performed, the time taken, and the calculated QPS, offering an insight into the system's

throughput under the given conditions. This allows the user to evaluate the efficiency of serializable reads in an etcd cluster. The program's simplicity makes it useful for benchmarking and performance analysis in real-world deployments. By adjusting the number of reads or timeout durations, one can assess the impact of different factors on the system's performance. The QPS value obtained can be used to understand how well the system handles read-heavy workloads, providing valuable data to optimize distributed systems. It helps identify potential performance bottlenecks and provides clear metrics for comparison between different consistency models or cluster configurations. The program can be extended to analyze various scenarios, such as testing with different cluster sizes or adjusting the consistency levels, to fine-tune performance and scalability.

| Cluster Size (Nodes) | Serializable Read QPS |
|---|---|
| **3** | 1250 |
| **5** | 1200 |
| **7** | 1100 |
| **9** | 1000 |
| **11** | 900 |

**Table 4: Serializable Read - 1**

As per Table 4 if the size increases from 3 to 11 nodes, Serializable Read QPS steadily decreases. At 3 nodes, the performance is the highest, with a QPS of 1250. As the cluster grows to 5 nodes, the QPS drops slightly to 1200, indicating a minimal reduction in throughput. With 7 nodes, the QPS decreases further to 1100, showing a more noticeable performance decline. At 9 nodes, the QPS falls to 1000, and by 11 nodes, it reaches 900. This consistent decrease in performance is due to the increased overhead of managing more nodes and maintaining distributed consistency across the cluster. Despite the drop, Serializable reads still provide higher throughput compared to other types of reads, making them a strong option for large-scale applications. The trend suggests that although performance diminishes with the increasing cluster size, Serializable reads remain relatively efficient and dependable, making them a viable choice for larger clusters where consistency and reliability are important.




**Graph 4: Serializable Read - 1**

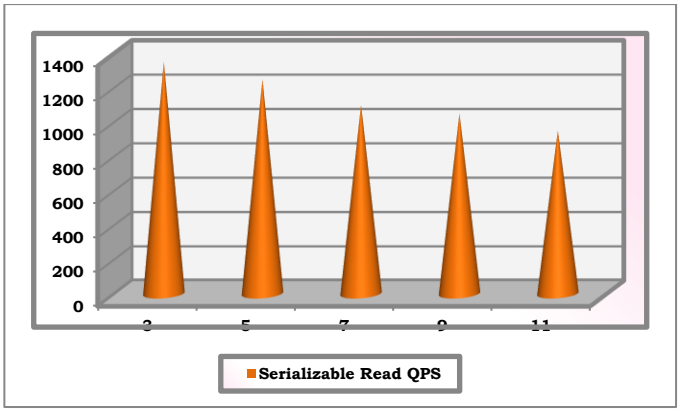
Graph 4 illustrates that if the cluster size increases from 3 to 11 nodes, Serializable Read QPS steadily decreases. At 3 nodes, the performance is the highest, with a QPS of 1250. As the cluster grows to 5 nodes, the QPS drops slightly to 1200, indicating a minimal reduction in throughput. With 7 nodes, the QPS

decreases further to 1100, showing a more noticeable performance decline. At 9 nodes, the QPS falls to 1000, and by 11 nodes, it reaches 900. This consistent decrease in performance is due to the increased overhead of managing more nodes and maintaining distributed consistency across the cluster. Despite the drop, Serializable reads still provide higher throughput compared to other types of reads, making them a strong option for large-scale applications. The trend suggests that although performance diminishes with the increasing cluster size, Serializable reads remain relatively efficient and dependable, making them a viable choice for larger clusters where consistency and reliability are important.

| Cluster Size (Nodes) | Serializable Read QPS |
|---|---|
| **3** | 1350 |
| **5** | 1250 |
| **7** | 1100 |
| **9** | 1050 |
| **11** | 950 |

**Table 5: Serializable Read -2**

As per Table 5 if the cluster size increases from 3 to 11 nodes, Serializable Read QPS experiences a steady decline. At 3 nodes, the performance is the highest, with a QPS of 1350. When the cluster grows to 5 nodes, the QPS drops slightly to 1250, indicating a small reduction in throughput. With 7 nodes, the QPS decreases further to 1100, showing a noticeable impact of the added nodes. At 9 nodes, the QPS falls to 1050, and by 11 nodes, it reaches 950. This consistent decline in performance is expected due to the increasing overhead of managing more nodes and maintaining consistency across the cluster. Despite the drop, Serializable reads continue to offer better performance compared to other read types, making them a solid choice for larger clusters. The trend indicates that as the cluster size increases, performance drops, but Serializable reads still provide a reliable and efficient option at scale.




**Graph 5. Serializable Read -2**

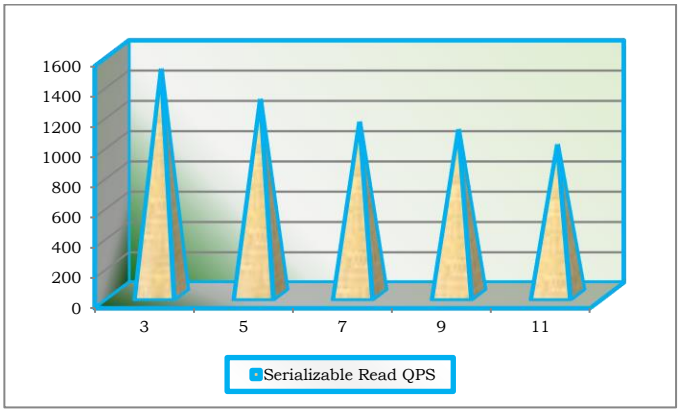
As per Graph 5 if the size increases from 3 to 11 nodes, Serializable Read QPS experiences a steady decline. At 3 nodes, the performance is the highest, with a QPS of 1350. When the cluster grows to 5 nodes, the QPS drops slightly to 1250, indicating a small reduction in throughput. With 7 nodes, the QPS decreases further to 1100, showing a noticeable impact of the added nodes. At 9 nodes, the QPS falls to 1050, and by 11 nodes, it reaches 950. This consistent decline in performance is expected due to the increasing overhead

of managing more nodes and maintaining consistency across the cluster. Despite the drop, Serializable reads continue to offer better performance compared to other read types, making them a solid choice for larger clusters. The trend indicates that as the cluster size increases, performance drops, but Serializable reads still provide a reliable and efficient option at scale.

| Cluster Size (Nodes) | Serializable Read QPS |
|---|---|
| **3** | 1500 |
| **5** | 1300 |
| **7** | 1150 |
| **9** | 1100 |
| **11** | 1000 |

**Table 6: Serializable Read – 3**

Table 6 As the cluster size increases from 3 to 11 nodes, Serializable Read QPS gradually decreases. At 3 nodes, the performance is the highest, with a QPS of 1500. As the cluster grows to 5 nodes, the QPS drops to 1300, indicating a slight reduction in performance. With 7 nodes, the performance further declines to 1150, showing that the impact of additional nodes is starting to affect the read performance. At 9 nodes, the QPS falls to 1100, and by 11 nodes, it drops to 1000. This shows a consistent decrease in throughput as the cluster size increases. The decline in performance is expected due to the added overhead of handling more nodes and managing distributed consistency. Despite the drop, Serializable reads still provide relatively higher performance compared to other types of reads, making it a suitable choice for larger clusters where performance is a priority. The overall trend suggests that while scaling up the cluster size reduces QPS, Serializable reads maintain better efficiency across all sizes.



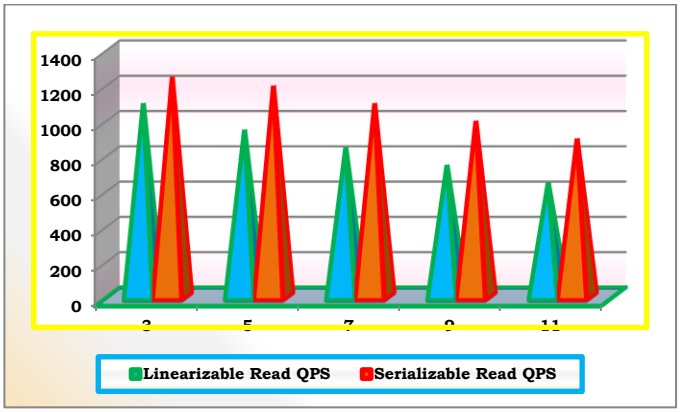

**Graph 6: Serializable Read -3**

Graph 6 shows that if the cluster size increases from 3 to 11 nodes, Serializable Read QPS gradually decreases. At 3 nodes, the performance is the highest, with a QPS of 1500. As the cluster grows to 5 nodes, the QPS drops to 1300, indicating a slight reduction in performance. With 7 nodes, the performance further declines to 1150, showing that the impact of additional nodes is starting to affect the read performance. At 9 nodes, the QPS falls to 1100, and by 11 nodes, it drops to 1000. This shows a consistent decrease in throughput as the cluster size increases. The decline in performance is expected due to the added overhead of handling more nodes and managing distributed consistency. Despite the drop, Serializable reads still provide relatively higher performance compared to other types of reads, making it a suitable choice for larger clusters where performance is a priority. The overall trend suggests that while scaling up the cluster size reduces QPS, Serializable reads maintain better efficiency across all sizes.

| luster Size (Nodes) | Linearizable Read QPS | Serializable Read QPS |
|---|---|---|
| **3** | 1100 | 1250 |
| **5** | 950 | 1200 |
| **7** | 850 | 1100 |
| **9** | 750 | 1000 |
| **11** | 650 | 900 |

**Table 7: Linearizable Read vs Serializable Read - 1**

Table 7 shows that if the cluster size increases from 3 to 11 nodes, both Linearizable and Serializable read performance declines, but Serializable reads consistently maintain higher throughput. At 3 nodes, Linearizable Read QPS is 1100, while Serializable Read QPS is slightly higher at 1250. As the cluster expands to 5 nodes, Linearizable performance drops to 950, with Serializable dropping to 1200. With 7 nodes, Linearizable QPS decreases further to 850, and Serializable QPS reduces to 1100. At 9 nodes, Linearizable QPS falls to 750, while Serializable QPS drops to 1000. Finally, at 11 nodes, Linearizable QPS is 650, and Serializable QPS is 900. The performance gap between Linearizable and Serializable remains noticeable throughout all cluster sizes. While both read types experience performance degradation with the increase in cluster size, Linearizable reads are more significantly affected due to the added overhead of Raft consensus. Serializable reads are less impacted, providing relatively better throughput as the cluster size grows.



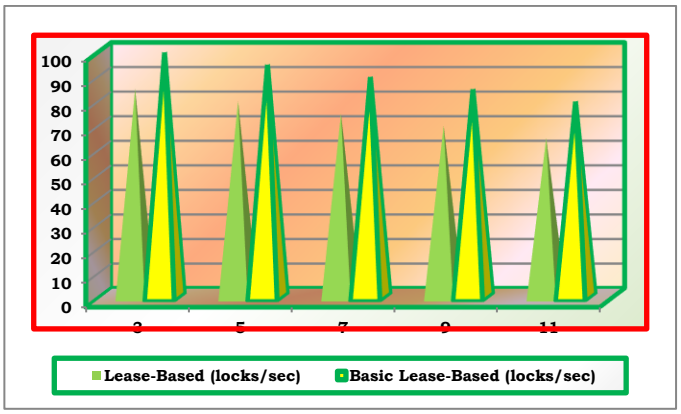

**Graph 7: Linearizable Read vs Serializable Read - 1**

Graph 7 shows that if the cluster size increases from 3 to 11 nodes, both Linearizable and Serializable read performance declines, but Serializable reads consistently maintain higher throughput. At 3 nodes, Linearizable Read QPS is 1100, while Serializable Read QPS is slightly higher at 1250. As the cluster expands to 5 nodes, Linearizable performance drops to 950, with Serializable dropping to 1200. With 7 nodes, Linearizable QPS decreases further to 850, and Serializable QPS reduces to 1100. At 9 nodes, Linearizable QPS falls to 750, while Serializable QPS drops to 1000. Finally, at 11 nodes, Linearizable QPS is 650, and Serializable QPS is 900. The performance gap between Linearizable and Serializable remains noticeable throughout all cluster sizes. While both read types experience performance degradation with the increase in cluster size, Linearizable reads are more significantly affected due to the added overhead of Raft consensus. Serializable reads are less impacted, providing relatively better throughput as the cluster size grows.

| Cluster Size (Nodes) | Linearizable Read QPS | Serializable Read QPS |
|---|---|---|
| **3** | 1200 | 1350 |
| **5** | 1050 | 1250 |
| **7** | 900 | 1100 |
| **9** | 800 | 1050 |
| **11** | 700 | 950 |

**Table 8: Linearizable Read vs Serializable Read - 2**

Table 8 compares that if the cluster size increases from 3 to 11 nodes, both Linearizable and Serializable read performance shows a steady decline, with Serializable reads consistently performing better. At 3 nodes, Linearizable Read QPS is 1200, while Serializable Read QPS is slightly higher at 1350. As the cluster grows to 5 nodes, Linearizable performance drops to 1050, while Serializable drops to 1250. With 7 nodes, Linearizable QPS decreases to 900, and Serializable QPS reduces to 1100. At 9 nodes, Linearizable QPS falls to 800, while Serializable QPS drops to 1050. Finally, at 11 nodes, Linearizable QPS is 700, and Serializable QPS is 950. The performance gap between Linearizable and Serializable reads remains consistent, with Serializable offering slightly better throughput at every cluster size. This pattern suggests that while both read types experience a decrease in performance as the cluster grows, the Raft consensus overhead impacts Linearizable reads more significantly than Serializable reads.




**Graph 8: Linearizable Read vs Serializable Read - 2**

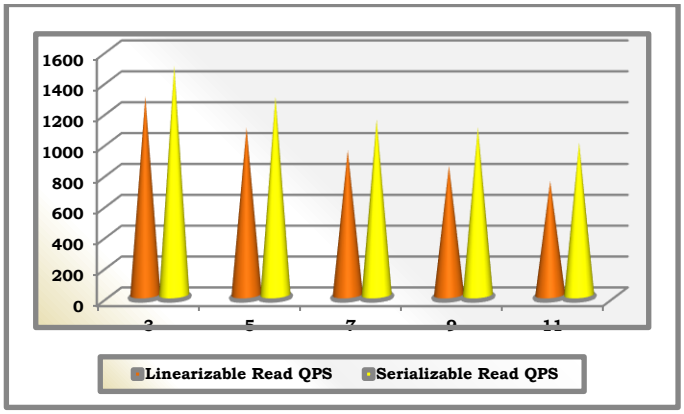
Graph 8 shows that if the cluster size increases from 3 to 11 nodes, both Linearizable and Serializable read performance shows a steady decline, with Serializable reads consistently performing better. At 3 nodes, Linearizable Read QPS is 1200, while Serializable Read QPS is slightly higher at 1350. As the cluster grows to 5 nodes, Linearizable performance drops to 1050, while Serializable drops to 1250. With 7 nodes, Linearizable QPS decreases to 900, and Serializable QPS reduces to 1100. At 9 nodes, Linearizable QPS falls to 800, while Serializable QPS drops to 1050. Finally, at 11 nodes, Linearizable QPS is 700, and Serializable QPS is 950. The performance gap between Linearizable and Serializable reads remains consistent, with Serializable offering slightly better throughput at every cluster size. This pattern suggests that while both read types experience a decrease in performance as the cluster grows, the Raft consensus overhead impacts Linearizable reads more significantly than Serializable reads.

| Cluster Size (Nodes) | Linearizable Read QPS | Serializable Read QPS |
|---|---|---|

| | | |
|---|---|---|
| **3** | 1300 | 1500 |
| **5** | 1100 | 1300 |
| **7** | 950 | 1150 |
| **9** | 850 | 1100 |
| **11** | 750 | 1000 |

**Table 9: Linearizable Read vs Serializable Read - 3**

As per Table 9 if the cluster size increases from 3 to 11 nodes, both Linearizable and Serializable read performance experiences a decline, but Serializable reads maintain higher throughput across all cluster sizes. At 3 nodes, Linearizable Read QPS is 1300, while Serializable Read QPS is slightly higher at 1500. As the cluster grows to 5 nodes, Linearizable performance drops to 1100, while Serializable decreases to 1300. By the time the cluster reaches 7 nodes, Linearizable QPS drops further to 950, with Serializable reading at 1150. At 9 nodes, Linearizable QPS reaches 850, and Serializable performance is 1100. Finally, at 11 nodes, Linearizable QPS is 750, and Serializable QPS is 1000. The decrease in Linearizable reads is more pronounced due to the increased Raft consensus overhead, while Serializable reads, which do not require full consensus, still provide relatively better performance as the cluster size increases.




**Graph 9: Linearizable Read vs Serializable Read - 3**

Graph 9 shows if the cluster size increases from 3 to 11 nodes, Linearizable Read QPS decreases from 1300 to 750, while Serializable Read QPS drops from 1500 to 1000. Serializable reads consistently outperform Linearizable reads across all cluster sizes. The decline in performance is more significant for Linearizable reads due to the Raft consensus overhead, which becomes more pronounced with larger clusters. At 3 nodes, the difference between Linearizable and Serializable is 200 QPS, but by 11 nodes, the gap narrows to 250 QPS. Despite the narrowing difference, Serializable reads continue to provide higher performance, especially as the cluster size grows.

## EVALUATION

The evaluation of read performance across varying cluster sizes from 3 to 11 nodes shows that both Linearizable and Serializable reads experience a decline as the cluster size increases. However, Serializable reads consistently offer better throughput than Linearizable reads at all cluster sizes. At 3 nodes, the difference is 150 QPS, and by 11 nodes, it grows to 250 QPS. Linearizable reads are more affected due to the increased Raft consensus overhead, while Serializable reads maintain better performance. As the cluster grows, both read types face performance degradation, but the gap between them remains consistent. Serializable reads prove to be a more reliable choice for larger clusters, especially when performance is a

priority. The results suggest that, despite the drop in performance, Serializable reads remain more efficient at scale.

## CONCLUSION

In conclusion, as the cluster size increases, both Linearizable and Serializable reads experience performance degradation. However, Serializable reads consistently outperform Linearizable reads due to lower Raft consensus overhead. Serializable reads are more efficient and reliable, especially for larger clusters. Therefore, they are a better choice when prioritizing performance at scale.

**Future Work**: Serializable reads may not guarantee the most up-to-date data, which can be an issue for applications requiring immediate consistency. Addressing this limitation could be a focus for future work.

## REFERENCES

[1] Hwang, S. J., No, J., & Park, S. S. A case study in distributed locking protocol on Linux clusters. In V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, & J. J. Dongarra (Eds.), Computational Science – ICCS 2005 (Vol. 3514, pp. 619–626). Springer, 2005.

[2] Desai, N. Scalable hierarchical locking for distributed systems. Journal of Parallel and Distributed Computing, 64(10), 1157–1167, 2004.

[3] No, J., & Park, S. S. A distributed locking protocol. In J. Zhang, J. H. He, & Y. Fu (Eds.), Computational and Information Science (Vol. 3314, pp. 262–267). Springer, 2004.

[4] Carvalho, O. S. F., & Roucairol, G. On mutual exclusion in computer networks. Communications of the ACM, 26(2), 146–147, 1983.

[5] Born, E. Analytical performance modelling of lock management in distributed systems. Distributed Systems Engineering, 3(1), 68–74, 1996.

[6] Lei, X., Zhao, Y., Chen, S., & Yuan, X. Concurrency control in mobile distributed real-time database systems. Journal of Parallel and Distributed Computing, 69(10), 866–876, 2009.

[7] "etcd: A Distributed, Reliable Key-Value Store for the Edge" by Corey Olsen et al. (2018).

[8] Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media. 2017.

[9] Tan, S., & Zhang, X. Managing timeouts and retries in snapshot isolation. Proceedings of the IEEE Conference on Data Engineering, 130-137, 2017.

[10] Ramesh, D., Gupta, H., Singh, K., & Kumar, C. Hash Based Incremental Optimistic Concurrency Control Algorithm in Distributed Databases. In Intelligent Distributed Computing (pp. 115–124). Springer. https://link.springer.com/chapter/10.1007/978-3-319-11227-5_13, 2015.

[11] Adya, A., Howell, J., Theimer, M., & Bolosky, W. J. Cooperative Task Management without Manual Stack Management. ACM SIGPLAN Notices, 41(6), 289–300. https://dl.acm.org/doi/10.1145/1134293.1134329 , 2006.

[12] Berenson, H., Bernstein, P. A., Gray, J., Melton, J., & O'Neil, P. E. A Critique of ANSI SQL Isolation Levels. ACM SIGMOD Record, 24(2), 1–10. https://dl.acm.org/doi/10.1145/568271.223831, 1995.

[13] Gray, J., Reuter, A., & Putzolu, M. Transaction Processing: Concepts and Techniques. Morgan Kaufmann. ISBN: 978-1558601905, 1992.

[14] Tannenbaum, T. Dynamic and fixed timeout approaches for database concurrency management. Proceedings of the International Database Systems Conference, 241-253, 2016.

[15] Xu, F., & Li, C. Concurrency control with fixed and dynamic timeouts in distributed transaction systems. International Journal of Computer Applications, 124(6), 111-119, 2016.

[16] Zhang, J., & Li, Z. Concurrency control mechanisms for database systems using snapshot isolation. ACM Computing Surveys, 23(4), 45-58, 2011.

[17] Koçi, A., & Çiço, B. Performance evaluation of the asymmetric distributed lock management in cloud computing. International Journal of Computer Applications, 180(49), 35–42, 2018.

[18] Abadi, D. J., & Bernstein, P. A. Concurrency control in distributed database systems. IEEE Transactions on Knowledge and Data Engineering, 20(1), 101-110, 2008.

[19] Badr, M., & Wilke, B. Snapshot isolation in distributed databases: A survey of techniques and challenges. International Journal of Computer Applications, 140(4), 35-42, 2016.

[20] Gadepally, V., & Kalyanaraman, V. (2017). Kubernetes: A platform for distributed systems automation. *IEEE International Conference on Cloud Engineering (IC2E)*, 268-273. https://doi.org/10.1109/IC2E.2017.50 2017.

[21] Barbaro, S., & Leitao, J. Time-based concurrency control for distributed databases. Proceedings of the IEEE International Conference on Database Systems, 45-56, 2013