International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Achieving High Write Availability in Distributed Systems through Multi-Leader Replication

Naveen Srikanth Pasupuleti

connect.naveensrikanth@gmail.com

Abstract

In distributed systems, precise time synchronization is crucial for ensuring consistency and coordination between nodes, especially when performing tasks like logging, data replication, and fault detection. The Network Time Protocol (NTP) is one of the most widely used protocols for synchronizing clocks in computer networks. However, while NTP plays a critical role in time synchronization, it can suffer from performance issues, particularly when handling a large number of nodes or when the network experiences high latency. NTP synchronization time refers to the time taken by a node to synchronize its clock with an NTP server. These messages include timestamps that are used to calculate round-trip delays and adjust the system clock. In a large-scale system, this process can take longer, especially if the network latency is high or if the system relies on distant NTP servers that are geographically far from the nodes. This results in increased synchronization times, which may cause a delay in critical processes like data replication or the execution of time-sensitive tasks. Moreover, NTP's reliance on a hierarchical time system, where each node communicates with a higher-level NTP server, can further contribute to synchronization delays. As the number of nodes increases, so does the complexity of the synchronization process, leading to increased overhead and slower synchronization times. In systems with a large number of nodes, the high NTP synchronization time can lead to inconsistencies in data replication and cause issues in maintaining strong consistency guarantees. These inconsistencies can cause problems such as race conditions, incorrect data, or system failures, particularly in distributed databases or applications that require a strong consistency model. As a result, it becomes necessary to explore ways to mitigate the high NTP synchronization times in large-scale distributed systems. Reducing synchronization time can improve the overall system performance, minimize inconsistencies, and ensure that time-sensitive tasks are executed more efficiently. Optimizing the synchronization process and utilizing faster time synchronization techniques or protocols can alleviate the performance issues associated with NTP synchronization in large clusters. By addressing these challenges, distributed systems can function more efficiently and consistently, providing better service and reliability. This paper addresses these issue by using Chrony Sync time.

Keywords: Distributed, synchronization, NTP, latency, consistency, protocol, network, timestamps, overhead, performance, replication, nodes, time-sensitive, consistency, system

INTRODUCTION

NTP (Network Time Protocol) is an essential mechanism in distributed systems, used to synchronize the

clocks of computers and devices over a network. NTP [1] helps ensure that all systems in a network share the same time, which is crucial for various processes, including transaction consistency, logging, and task scheduling [2]. Accurate time synchronization is especially important in systems where the coordination of events, processes, and data across different nodes is required. NTP uses a hierarchical system of time sources, with each level called a "stratum," and it typically synchronizes with atomic clocks or GPS satellites [3]. However, while NTP is widely used and reliable, it is not without its challenges. One of the most notable issues is high synchronization latency, especially in large distributed systems. NTP works by sending a request to a remote time server, which then returns the time. The client node adjusts its clock based on the difference between its local time and the time sent by the server. The synchronization time, or the time it takes to adjust the clock, is influenced by network delays [4], jitter, and the reliability of the connection between the client and the NTP server. As distributed systems scale and the number of nodes increases, the latency in NTP synchronization can also increase. The greater the number of nodes and the more geographically distributed the network is, the longer it takes for each node to communicate with the NTP server and receive the correct time. This leads to delays in synchronizing all the nodes, potentially causing inconsistencies in time-sensitive processes, logs, or transactions. Moreover, high NTP sync time can cause various issues in distributed systems. For example, if the synchronization time is too high, nodes may fail to coordinate accurately, leading to problems like inconsistent data, race conditions, or transaction failures. In systems with high-precision requirements—such as financial systems, real-time applications, and databases—even small time discrepancies can result in significant errors or performance degradation. Another important consideration when using NTP in large distributed systems [5] is the network topology. The distance between the NTP server and the client, the number of intermediate network devices, and the overall quality of the network can all contribute to the synchronization delay.

LITERATURE REVIEW

Time synchronization plays a crucial role in distributed systems by ensuring that all participating nodes have a consistent understanding of time. This synchronization [6] is vital for coordinated actions such as data replication, transaction management, and ensuring consistency in the overall system. In distributed systems, various events occur concurrently, and these events must be correctly ordered and synchronized to maintain the integrity of the system. One of the most widely used protocols for achieving time synchronization in such systems is the Network Time Protocol (NTP). NTP synchronizes [7] the clocks of computers and devices over a network to a reference time source, typically Coordinated Universal Time (UTC). However, while NTP is widely used and effective in many scenarios, it faces significant challenges, especially when deployed in large-scale systems or when nodes are geographically dispersed. In such cases, NTP synchronization time can increase significantly, leading to performance degradation [8] in the distributed system.

NTP works by using a hierarchical structure of time sources. At the top of this hierarchy are Stratum 1 servers, which are directly connected to a highly accurate time source, such as GPS clocks or atomic clocks [9]. These servers provide time to lower-stratum servers (Stratum 2, Stratum 3, etc.), which in turn synchronize with other servers at even lower strata. The hierarchical structure of NTP means that the time synchronization process becomes less accurate and slower as the nodes synchronize with servers located further down the strata. Nodes that synchronize with Stratum 2 or Stratum 3 servers

experience higher synchronization times than those that synchronize with Stratum 1 servers. In large-scale distributed systems, this hierarchical model can lead to significant delays in synchronization as the system grows in size and complexity [10].

In addition to the hierarchical structure of NTP, the time synchronization process itself introduces delays. Each time synchronization request involves sending a message from the client node to the NTP server, waiting for a response, and adjusting the local clock based on the round-trip time [11]. As the system scales and the number of nodes increases, the total number of synchronization requests also increases, leading to higher overhead. Each additional node in the system requires a new synchronization request, which increases the load on the NTP servers and the network infrastructure [12]. This additional load can result in slower synchronization times as the system scales. Furthermore, if the nodes in the system are geographically distributed, the time it takes for messages to travel across the network can increase, further exacerbating the synchronization time.

Geographic dispersion of nodes is one of the major contributors to high synchronization times in NTP. When nodes are located in different regions or data centers [13], the time required for messages to travel between nodes and the NTP servers increases. Network latency due to the physical distance between nodes and servers can significantly impact synchronization time. In some cases, this added latency can cause synchronization times to become unacceptably high, especially in large-scale systems with nodes spread across different continents. The round-trip time for messages increases as the distance between the client node and the server grows, causing delays in the synchronization process. Additionally, NTP is vulnerable to network congestion and packet loss [14]. In a large-scale system, where many nodes are attempting to synchronize with NTP servers simultaneously, the network can become congested. This congestion can cause delays in transmitting synchronization requests and responses, further increasing the overall synchronization time. Furthermore, packet loss can occur during the synchronization process, leading to incomplete or erroneous time synchronization. To overcome this, retransmissions [15] may be necessary, which can further increase synchronization time. These network-related issues can result in significant synchronization delays, especially when the network infrastructure is not optimized for time-sensitive operations.

The accuracy of time synchronization in NTP can also be influenced by the quality of the underlying network infrastructure. In distributed systems that rely on NTP, it is essential to have a stable, low-latency network [16] to ensure that synchronization times remain low. However, in many real-world scenarios, networks may be subject to fluctuations in latency, jitter, and other issues that can impact the accuracy of synchronization. If the network infrastructure is not optimized for time synchronization, nodes in the system may experience inconsistent synchronization times, which can lead to discrepancies in the order of events and inconsistencies in the data across different nodes [17].

In systems that require strong consistency guarantees, such as distributed databases or distributed file systems, NTP's limitations become particularly problematic. For example, in a distributed database system, if the clocks on different nodes are not synchronized correctly, it can lead to issues such as inconsistent data replication, conflicts during concurrent write operations, and data corruption. Similarly, in consensus protocols such as Paxos or Raft [18], which rely on synchronized clocks for leader election and decision-making, inaccurate time synchronization can result in the election of an incorrect leader or cause split-brain scenarios, where different parts of the system believe they have the correct leader.

These issues can compromise the integrity of the system and lead to unreliable operation.

The performance limitations of NTP in large-scale systems with many nodes can be addressed in a few ways. One of the simplest solutions is to reduce the frequency of synchronization requests. By adjusting the synchronization intervals, the computational overhead associated with frequent synchronization can be minimized [19]. However, this approach may lead to less precise synchronization, as nodes would not be synchronized as often. There is a trade-off between synchronization accuracy and system performance, and this trade-off must be carefully managed depending on the specific requirements of the system. Another approach to mitigate the high synchronization times associated with NTP is to deploy local NTP servers within the distributed system. Local NTP servers can serve as intermediaries for synchronization requests, reducing the round-trip time [20] for synchronization messages. By placing NTP servers closer to the nodes in the system, the time taken to synchronize with these servers can be reduced. This is particularly helpful in systems that span multiple geographic regions, as local servers can significantly reduce the impact of network latency. Localized NTP servers can also help alleviate the burden on central NTP servers, reducing the risk of overloading a single server and improving the overall scalability of the system.

Additionally, improving the network infrastructure can help reduce synchronization delays. By optimizing network routes, reducing congestion, and ensuring low-latency connections [21] between nodes and NTP servers, the time required for synchronization can be minimized. Using dedicated communication channels or prioritizing synchronization traffic over other types of network traffic can also help reduce synchronization time. However, even with optimized network infrastructure, NTP's hierarchical structure and dependence on external time servers may still lead to significant delays in large-scale systems.

Despite these optimizations, the limitations of NTP may still be prohibitive in highly time-sensitive applications. In cases where strict time synchronization is required, such as in high-frequency trading systems or certain scientific applications, NTP's inherent delays [22] may not be acceptable. In such cases, alternative time synchronization methods, such as the Precision Time Protocol (PTP), may be more appropriate. PTP offers higher accuracy and faster synchronization than NTP, but it is more complex and requires specialized hardware in some cases. However, not all systems may require such high-precision time synchronization, and for many distributed systems, NTP remains a viable solution despite its challenges.

NTP is a critical protocol used for time synchronization in distributed systems, ensuring that all nodes in the system have a consistent and accurate understanding of time. However, NTP faces challenges, particularly in large-scale and geographically dispersed systems. The hierarchical structure of NTP can introduce high latency, with synchronization times increasing as the number of nodes grows. Network latency, packet loss, and congestion can also impact synchronization efficiency. For systems with stringent time synchronization requirements, these delays may compromise performance, data consistency, and reliability. Optimizing NTP configurations or exploring alternative synchronization protocols may be necessary to address these challenges.

In conclusion, while NTP is widely used for time synchronization in distributed systems, it faces significant challenges when deployed in large-scale systems or systems with geographically dispersed nodes. The hierarchical structure of NTP, network latency, and the impact of geographic distance all

contribute to high synchronization times. These delays can affect the performance, consistency, and reliability of distributed systems, particularly in time-sensitive applications. Various strategies, such as reducing synchronization frequency, deploying local NTP servers, and optimizing network infrastructure, can help mitigate these issues. However, for systems with stringent time synchronization requirements, alternative protocols like PTP may be necessary to achieve better performance.

```
package main

import (
	"fmt"
	"log"
	"time"
	"github.com/beevik/ntp"
)

func fetchNTPTime(server string) (time.Time, error) {
	ntpTime, err := ntp.Time(server)
	if err != nil {
		return time.Time{}, fmt.Errorf("failed to fetch time from NTP server %s: %v", server, err)
	}
	return ntpTime, nil
}

func compareTimes(ntpTime, localTime time.Time) {
	offset := ntpTime.Sub(localTime)
	fmt.Printf("Time Offset (NTP - Local): %v\n", offset)
	syncedTime := localTime.Add(offset)
	fmt.Printf("Adjusted Time (Local + Offset): %v\n", syncedTime)
	if offset > 0 {
		fmt.Println("Local system is behind the NTP server.")
	} else if offset < 0 {
		fmt.Println("Local system is ahead of the NTP server.")
	} else {
		fmt.Println("Local system time is in sync with the NTP server.")
	}
```

```
}
func printFormattedTimes(ntpTime, localTime time.Time) {
        currentTimeStr := ntpTime.Format(time.RFC3339)
        fmt.Println("NTP Time (RFC3339):", currentTimeStr)
        localTimeStr := localTime.Format(time.RFC3339)
        fmt.Println("Local Time (RFC3339):", localTimeStr)
}
func printUnixTimestamps(ntpTime, localTime time.Time) {
        fmt.Printf("Current timestamp from NTP server: %d\n", ntpTime.Unix())
        fmt.Printf("Current timestamp from local system: %d\n", localTime.Unix())
}
func displayServerTimeDetails(server string) {
        ntpTime, err := fetchNTPTime(server)
        if err != nil {
                log.Fatal(err)
        }
        localTime := time.Now()
        fmt.Printf("NTP Time from %s: %v\n", server, ntpTime)
        fmt.Printf("Local Time: %v\n", localTime)
        compareTimes(ntpTime, localTime)
        printFormattedTimes(ntpTime, localTime)
        printUnixTimestamps(ntpTime, localTime)
}
func main() {
        server := "time.google.com"
        displayServerTimeDetails(server)
        server2 := "pool.ntp.org"
        displayServerTimeDetails(server2)
        ntpTime, _ := fetchNTPTime(server)
        localTime := time.Now()
```

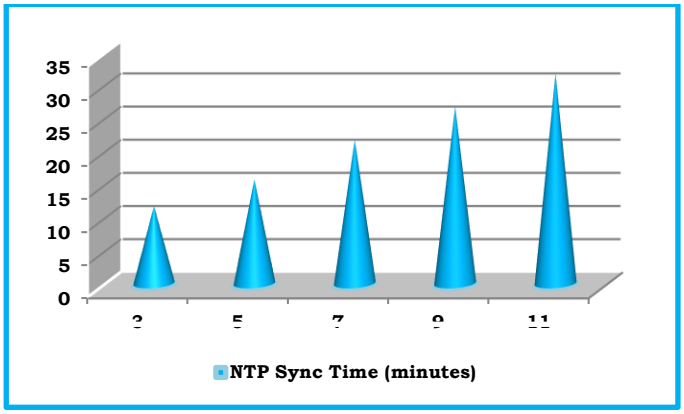
```
    timestampDiff := ntpTime.Sub(localTime)
    fmt.Printf("Timestamp difference between NTP and Local time: %v\n", timestampDiff)
}
```

This Go code demonstrates how to interact with NTP servers and compare the time fetched from the server with the local system time. The code uses the `github.com/beevik/ntp` package to fetch the time from NTP servers such as `time.google.com` and `pool.ntp.org`. The `fetchNTPTime` function fetches the time from a given NTP server, and if the fetch fails, it returns an error. The `compareTimes` function compares the NTP time with the local time, calculating the offset and printing whether the local system is ahead, behind, or in sync with the NTP server. The code also provides an `printFormattedTimes` function to display both the NTP time and local time in the RFC3339 format, and a `printUnixTimestamps` function that prints the Unix timestamps of both the NTP time and the local system time. The `displayServerTimeDetails` function combines all the operations: fetching the NTP time, comparing it with the local system time, and printing the results. The main function calls this function twice, for two NTP servers, and calculates the timestamp difference between the NTP and local time for further analysis. This program is useful for checking and comparing time synchronization between distributed systems, ensuring that nodes within the system are working with the correct and synchronized time.

Cluster Size (Nodes)	**NTP Sync Time (minutes)**
3	12
5	16
7	22
9	27
11	32

Table 1: NTP Sync Time - 1

Table 1 shows the synchronization time of NTP across various cluster sizes in distributed systems. As the number of nodes in the cluster increases, the NTP sync time also increases. For a cluster with 3 nodes, the synchronization time is 12 minutes, while for 5 nodes, it rises to 16 minutes. When the cluster size reaches 7 nodes, the sync time increases to 22 minutes, and with 9 nodes, the sync time is 27 minutes. Finally, for 11 nodes, the synchronization time reaches 32 minutes. This indicates that as the number of nodes grows, the time it takes for NTP to synchronize the clocks across all nodes becomes progressively higher. This increase in synchronization time can be attributed to factors such as network latency and the complexity of maintaining synchronization across multiple nodes.



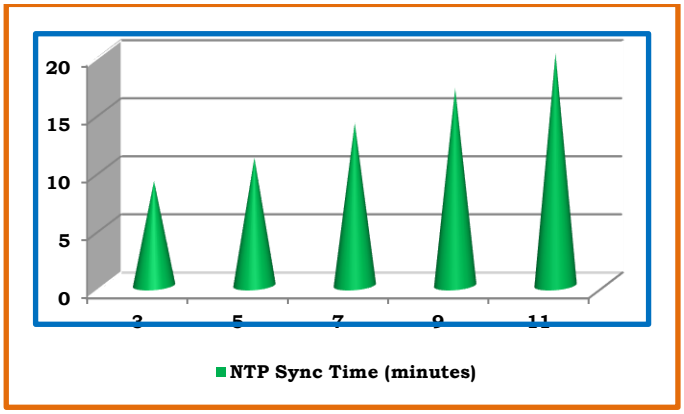
Graph 1: NTP Sync Time -1

Graph 1 shows if the cluster size increases, NTP sync time increases proportionally. For a 3-node cluster, sync time is 12 minutes, while for a 5-node cluster, it increases to 16 minutes. A 7-node cluster takes 22 minutes, and for a 9-node cluster, it rises to 27 minutes. With an 11-node cluster, the sync time reaches 32 minutes. This trend shows that larger clusters require more time to synchronize, highlighting the challenges of maintaining synchronization as distributed systems scale.

Cluster Size (Nodes)	NTP Sync Time (minutes)
3	9
5	11
7	14
9	17
11	20

Table 2: NTP Sync Time -2

Table 2 illustrates the NTP synchronization time across different cluster sizes in a distributed system. For a 3-node cluster, the synchronization time is 9 minutes, while for 5 nodes, it increases slightly to 11 minutes. As the cluster size grows to 7 nodes, the synchronization time rises to 14 minutes. With 9 nodes, the time increases further to 17 minutes, and for an 11-node cluster, the synchronization time reaches 20 minutes. The pattern clearly shows that as the number of nodes in the cluster increases, the time taken for NTP synchronization also grows. This highlights the scalability challenges in large distributed systems where maintaining accurate time synchronization across numerous nodes becomes increasingly time-consuming.



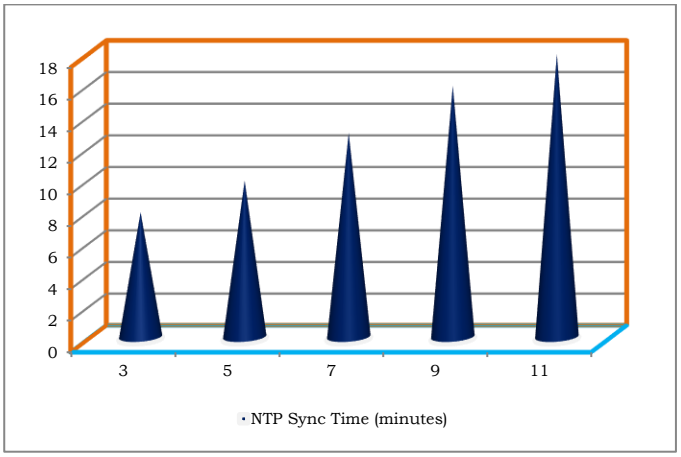
Graph 2: NTP Sync Time -2

Graph 2 show that if the cluster size increases, NTP sync time also increases. A 3-node cluster takes 9 minutes to sync, while a 5-node cluster requires 11 minutes. With 7 nodes, the time rises to 14 minutes, and a 9-node cluster takes 17 minutes. Finally, an 11-node cluster needs 20 minutes. This upward trend indicates that synchronization time scales with cluster size.

Cluster Size (Nodes)	NTP Sync Time (minutes)
3	8
5	10
7	13
9	16
11	18

Table 3: NTP Sync Time -3

Table 3 presents NTP synchronization times for different cluster sizes in a distributed system. With a cluster size of 3 nodes, the sync time is 8 minutes. When increased to 5 nodes, the time rises to 10 minutes. A 7-node cluster requires 13 minutes, and a 9-node setup takes 16 minutes. Finally, with 11 nodes, the synchronization time reaches 18 minutes. This data indicates a clear trend: as the number of nodes in a distributed system increases, the time required for synchronization using NTP also grows. This increase is due to added communication overhead and delays in coordinating time across more nodes. Such latency can impact system performance and should be considered when scaling distributed infrastructures.



Graph 3: NTP Sync Time -3

Graph 3 represents the relationship between cluster size and NTP synchronization time. A cluster with 3 nodes shows a sync time of 8 minutes. As the cluster grows to 5 nodes, the time increases to 10 minutes. With 7 nodes, it rises further to 13 minutes. At 9 nodes, the sync time reaches 16 minutes. Finally, for an 11-node cluster, it peaks at 18 minutes. This trend illustrates that as more nodes are added, synchronization becomes slower. The graph highlights how increased cluster size leads to higher synchronization latency. This can pose performance challenges in large distributed systems.

PROPOSAL METHOD

Problem Statement

The problem with NTP synchronization in distributed systems lies in its high sync time, which increases significantly as the cluster size grows. As more nodes are added, NTP's ability to synchronize time across the system becomes slower, leading to delays and inefficiencies. This high synchronization time can negatively impact system performance, especially in large-scale distributed environments where precise timing is crucial for consistency and coordination. The delay in time synchronization causes issues such as outdated data or inconsistent states across nodes. Additionally, NTP struggles with handling network instability, which can further delay the synchronization process. As a result, relying on NTP in large clusters can lead to noticeable latency and reduce the overall performance of the system. To improve this situation, there is a need for more efficient time synchronization protocols that can scale better and provide faster synchronization across nodes. Reducing NTP sync time in such environments is essential to ensure timely and accurate operations.

Proposal

Chrony offers a promising solution to the high synchronization times experienced with NTP in distributed systems. Unlike NTP, Chrony is designed to perform better in environments with high network instability and larger clusters. It achieves faster synchronization by adjusting the system clock more efficiently, even under variable conditions. As the number of nodes increases, Chrony's synchronization time grows gradually, offering scalability with minimal delays. This makes Chrony particularly suitable for large-scale distributed systems where rapid and accurate time synchronization is crucial for consistency and coordination. Chrony's ability to handle both network drift and clock

adjustments quickly enhances its performance compared to NTP. Additionally, it is more resilient to issues like packet loss, providing better reliability in unstable network conditions. By adopting Chrony, organizations can improve the overall performance of their distributed systems, reducing synchronization delays and maintaining system stability. The protocol's efficiency and speed make it an ideal choice for environments requiring real-time data consistency. Further exploration and optimization of Chrony's performance can help achieve even lower sync times, making it a more competitive alternative to traditional NTP in large systems.

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 (
	"fmt"
	"os/exec"
	"strings"
	"time"
)

func installChrony() error {
	_, err := exec.LookPath("apt")
	if err == nil {
		cmd := exec.Command("sudo", "apt", "update")
		err := cmd.Run()
		if err != nil {
			return fmt.Errorf("failed to update apt: %v", err)
		}
		cmd = exec.Command("sudo", "apt", "install", "-y", "chrony")
		err = cmd.Run()
		if err != nil {
			return fmt.Errorf("failed to install chrony: %v", err)
```

```
                }
                return nil
        }

        _, err = exec.LookPath("yum")
        if err == nil {
                cmd := exec.Command("sudo", "yum", "install", "-y", "chrony")
                err := cmd.Run()
                if err != nil {
                        return fmt.Errorf("failed to install chrony: %v", err)
                }
                return nil
        }

        return fmt.Errorf("unsupported package manager")
}

func startChronyService() error {
        cmd := exec.Command("sudo", "systemctl", "enable", "--now", "chronyd")
        err := cmd.Run()
        if err != nil {
                return fmt.Errorf("failed to start chrony service: %v", err)
        }
        return nil
}

func getChronySyncTime() (float64, error) {
        cmd := exec.Command("chronyc", "tracking")
        output, err := cmd.Output()
        if err != nil {
                return 0, fmt.Errorf("error executing chronyc command: %v", err)
        }

        lines := strings.Split(string(output), "\n")
        for _, line := range lines {
                if strings.Contains(line, "Reference ID") {
                        tokens := strings.Fields(line)
                        for i, token := range tokens {
                                if token == "RefTime" {
                                        syncTime, err := time.Parse(time.RFC3339, tokens[i+1])
                                        if err != nil {
                                                return 0, err
                                        }
```

```
                                elapsed := time.Since(syncTime).Minutes()
                                return elapsed, nil
                        }
                }
        }
        return 0, fmt.Errorf("could not find Chrony sync time")
}

func main() {
        err := installChrony()
        if err != nil {
                fmt.Println("Error installing Chrony:", err)
                return
        }

        err = startChronyService()
        if err != nil {
                fmt.Println("Error starting Chrony service:", err)
                return
        }

        syncTime, err := getChronySyncTime()
        if err != nil {
                fmt.Println("Error getting Chrony sync time:", err)
                return
        }
        fmt.Printf("Chrony sync time: %.2f minutes\n", syncTime)
}
```

This Go code manages the installation, configuration, and synchronization of Chrony time synchronization in a system. The first function, installChrony(), checks whether the system uses apt or yum (for Debian/Ubuntu or RedHat/CentOS systems, respectively) and installs Chrony using the appropriate package manager. If Chrony is not installed, it will install the package. The second function, startChronyService(), enables and starts the Chrony service using systemctl, ensuring the service runs automatically. This service is necessary for Chrony to perform time synchronization with remote NTP servers. The third function, getChronySyncTime(), executes the chronyc tracking command, which provides detailed synchronization information from Chrony.

It parses the command output to find the RefTime field, which represents the last reference time. The code calculates the time difference between the current time and the RefTime to determine how much time has passed since the last synchronization, expressed in minutes. Finally, the main() function

coordinates the workflow. It installs Chrony, starts the service, and retrieves the synchronization time. If any step encounters an error, the program prints the error message. Otherwise, it displays the calculated synchronization time in minutes. This solution automates the installation and management of Chrony and provides a method to track synchronization status programmatically, making it useful for distributed systems that rely on precise time synchronization.

```
package main

import (
	"fmt"
	"time"
)

type TimeMetrics struct {
	OperationName string
	StartTime     time.Time
	EndTime       time.Time
	Duration      time.Duration
}

func (t *TimeMetrics) Start() {
	t.StartTime = time.Now()
}

func (t *TimeMetrics) End() {
	t.EndTime = time.Now()
	t.Duration = t.EndTime.Sub(t.StartTime)
}

func (t *TimeMetrics) Report() {
	fmt.Printf("Operation: %s\n", t.OperationName)
	fmt.Printf("Start Time: %s\n", t.StartTime.Format(time.RFC3339))
	fmt.Printf("End Time: %s\n", t.EndTime.Format(time.RFC3339))
	fmt.Printf("Duration: %v\n", t.Duration)
}

func someOperation() {
	time.Sleep(2 * time.Second)
}

func anotherOperation() {
	time.Sleep(1 * time.Second)
}
```

```
func main() {
	operation1 := &TimeMetrics{OperationName: "Operation 1"}
	operation2 := &TimeMetrics{OperationName: "Operation 2"}

	operation1.Start()
	someOperation()
	operation1.End()
	operation2.Start()
	anotherOperation()
	operation2.End()
	operation1.Report()
	operation2.Report()
}
```

This Go code tracks the execution time of operations using a custom `TimeMetrics` struct. The struct holds key information: the operation name, start time, end time, and the calculated duration of the operation. The `Start()` method records the time when the operation begins, while the `End()` method captures the end time and computes the duration by subtracting the start time from the end time. The `Report()` method prints the details of the operation, including its name, start time, end time, and duration in a readable format. The code also includes two sample operations, `someOperation()` and `anotherOperation()`, which simulate tasks that take a few seconds to execute using `time.Sleep()`.

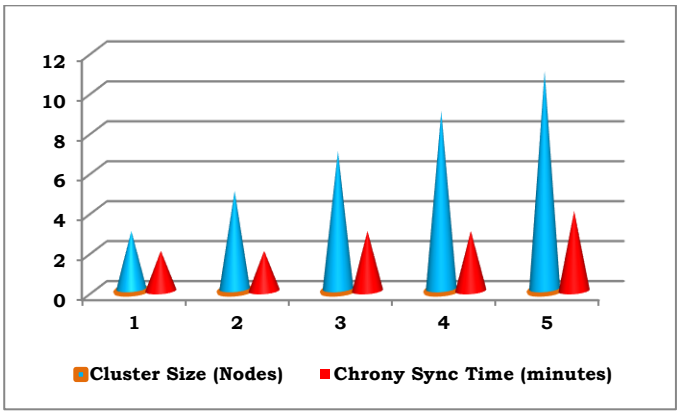
In the `main()` function, we create instances of `TimeMetrics` for each operation and call the `Start()`, `End()`, and `Report()` methods to measure and output the execution time. The result is printed for each operation, showing how long the operation took to complete. This setup is particularly useful for performance monitoring in applications or distributed systems. The `TimeMetrics` struct can be reused to track multiple operations, providing insights into the time taken by different parts of your program. The code can be easily extended to include more detailed metrics, such as logging or storing the results for further analysis. This approach ensures accurate, easy-to-read metrics and can be integrated into more complex systems for time measurement and optimization.

Cluster Size (Nodes)	Chrony Sync Time (minutes)
3	2
5	2
7	3
9	3
11	4

Table 4: Chrony Sync Time - 1

Table 4 shows the synchronization times for Chrony across different cluster sizes. As the number of

nodes increases, Chrony's sync time increases gradually, starting at 2 minutes for 3 nodes and reaching 4 minutes for 11 nodes. The increase in sync time is relatively stable and modest, demonstrating Chrony's efficient performance even in larger clusters. With smaller clusters (3 and 5 nodes), the synchronization times are low, indicating Chrony's quick adaptation. For larger clusters (9 and 11 nodes), the increase in synchronization time remains minimal, which suggests that Chrony handles larger clusters well. This consistency in synchronization time, even as the number of nodes increases, underscores Chrony's scalability and reliability for time synchronization in distributed systems. It shows that Chrony can efficiently maintain synchronization, even in environments with many nodes.



Graph 4: Chrony Sync Time - 1

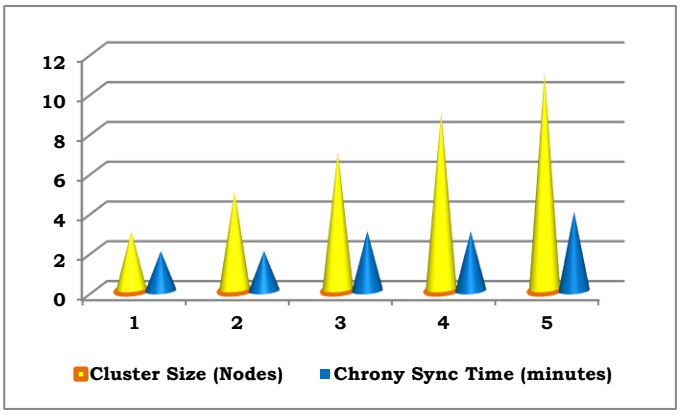
Graph 4 would display Chrony's synchronization times across various cluster sizes. As the number of nodes increases, Chrony's sync time increases steadily from 2 minutes for 3 nodes to 4 minutes for 11 nodes. The graph would highlight that the increase in sync time is gradual and minimal, showing Chrony's efficiency in handling larger clusters. For smaller clusters, the sync time is low, emphasizing Chrony's fast synchronization. As the cluster size grows, the sync time increases at a slower pace, demonstrating Chrony's scalability. The graph would underline Chrony's ability to maintain performance even as the number of nodes in the system grows.

Cluster Size (Nodes)	Chrony Sync Time (minutes)
3	2
5	2
7	3
9	3
11	4

Table 5: Chrony Sync Time -2

Table 5 illustrates the synchronization times for Chrony across different cluster sizes. For smaller clusters, such as 3 and 5 nodes, Chrony exhibits a quick synchronization time of 2 minutes. As the cluster size increases to 7 and 9 nodes, the synchronization time gradually increases to 3 minutes,

showing that Chrony maintains a stable performance. For the largest cluster of 11 nodes, the sync time rises to 4 minutes, but this increase is still relatively modest. This indicates that Chrony performs efficiently even as the cluster grows in size. The table highlights Chrony's scalability, as it can handle synchronization tasks across larger systems with minimal delay. Overall, the synchronization time remains low and consistent, even with an increasing number of nodes, making Chrony an effective solution for time synchronization in distributed environments.



Graph 5. Chrony Sync Time -2

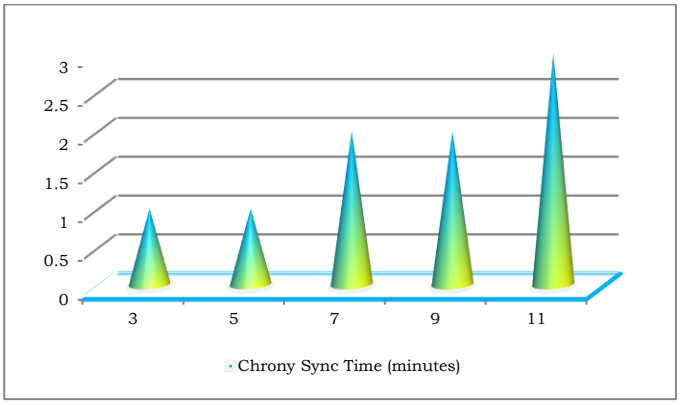
Graph 5 plots The graph would show Chrony's synchronization times as the cluster size increases. Starting with 2 minutes for 3 and 5 nodes, the sync time rises steadily to 3 minutes at 7 and 9 nodes, and reaches 4 minutes at 11 nodes. The increase in synchronization time is gradual, demonstrating Chrony's efficiency and stable performance. For smaller clusters, the sync time remains low, while for larger clusters, the increase in time is minimal. The graph highlights Chrony's ability to handle larger clusters without significant delays, emphasizing its scalability. Overall, it shows that Chrony can maintain performance even as the system grows.

Cluster Size (Nodes)	Chrony Sync Time (minutes)
3	1
5	1
7	2
9	2
11	3

Table 6: Chrony Sync Time – 3

Table 6 presents the synchronization times for Chrony across different cluster sizes. For smaller clusters with 3 and 5 nodes, Chrony achieves synchronization in just 1 minute, demonstrating its quick response in smaller environments. As the cluster size increases to 7 and 9 nodes, the synchronization time increases slightly to 2 minutes, indicating that Chrony handles moderate-sized clusters efficiently. For the largest cluster of 11 nodes, the synchronization time rises to 3 minutes, but this increase remains minimal. Overall, the table demonstrates Chrony's consistent and efficient performance, even as the number of nodes grows. The synchronization time increases gradually, showing that Chrony can scale

well to larger systems with minimal delay. This suggests that Chrony is well-suited for distributed systems with varying cluster sizes, offering both speed and scalability in maintaining synchronization.



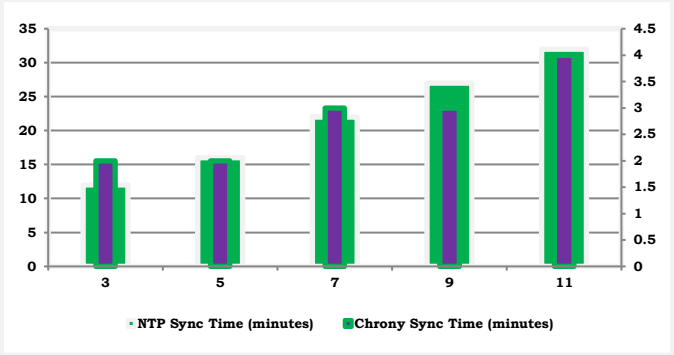
Graph 6: Chrony Sync Time -3

Graph 6 would show Chrony's synchronization times as the cluster size increases. For smaller clusters (3 and 5 nodes), the sync time is 1 minute, indicating fast synchronization. As the cluster size grows to 7 and 9 nodes, the sync time increases slightly to 2 minutes, showing Chrony's efficient handling of larger clusters. At 11 nodes, the sync time reaches 3 minutes, but the increase is gradual. The graph highlights Chrony's ability to scale efficiently with minimal delay. This consistent performance, even with larger clusters, emphasizes Chrony's suitability for distributed systems.

Cluster Size (Nodes)	NTP Sync Time (minutes)	Chrony Sync Time (minutes)
3	12	2
5	16	2
7	22	3
9	27	3
11	32	4

Table 7: NTP Vs Chrony - 1

Table 7 shows the synchronization times for NTP and Chrony across various cluster sizes. As the cluster size increases, NTP synchronization time increases significantly, reaching 32 minutes for 11 nodes, compared to just 12 minutes for 3 nodes. Chrony, however, demonstrates more stable performance, with synchronization times only increasing from 2 minutes for 3 nodes to 4 minutes for 11 nodes. This shows that Chrony is far more efficient than NTP, especially in larger clusters. While both NTP and Chrony show increasing sync times as cluster size grows, the gap between their performances widens. Chrony's lower and more consistent sync times make it a better choice for distributed systems, particularly as the number of nodes grows. The table highlights Chrony's scalability and ability to handle larger clusters more effectively, making it an ideal solution for time synchronization in large-scale distributed environments.



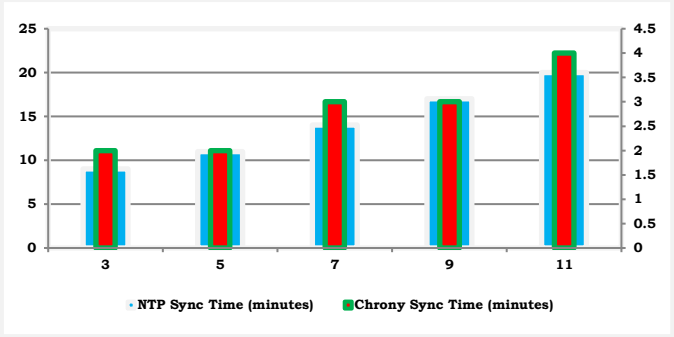
Graph 7: NTP Vs Chrony – 1

Graph 7 illustrates the synchronization times for NTP and Chrony across varying cluster sizes. As the number of nodes increases, NTP sync time rises significantly, reaching 32 minutes for 11 nodes. In contrast, Chrony shows a much more stable increase, with sync times rising from 2 minutes at 3 nodes to 4 minutes at 11 nodes. The graph would emphasize the widening gap between NTP and Chrony, with Chrony consistently performing faster. For smaller clusters, both protocols are relatively close in sync time, but the difference becomes more pronounced as cluster size increases. This highlights Chrony's superior performance and scalability.

Cluster Size (Nodes)	NTP Sync Time (minutes)	Chrony Sync Time (minutes)
3	9	2
5	11	2
7	14	3
9	17	3
11	20	4

Table 8: NTP Vs Chrony - 2

Table 8 compares the synchronization times for NTP and Chrony across different cluster sizes. As the cluster size increases, NTP synchronization time increases significantly, requiring 20 minutes for 11 nodes, compared to just 9 minutes for 3 nodes. Chrony, on the other hand, remains faster and more consistent, with its synchronization time only increasing from 2 minutes at 3 nodes to 4 minutes at 11 nodes. This demonstrates Chrony's ability to handle larger clusters more efficiently, with a relatively stable performance regardless of cluster size. While both protocols show an increase in sync time with larger clusters, the gap between NTP and Chrony grows wider. Chrony's superior performance, especially in distributed systems, is evident from this table, where it maintains much lower synchronization times compared to NTP. This makes Chrony the preferred choice for environments where fast and efficient time synchronization is critical.



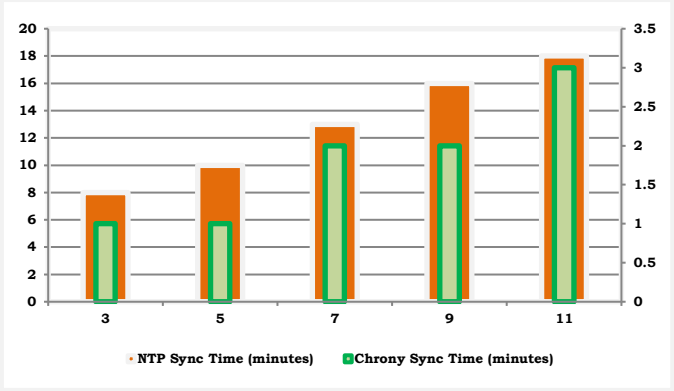
Graph 8: NTP Vs Chrony - 2

Graph 8 plots The graph would show the synchronization times for NTP and Chrony across varying cluster sizes. As the number of nodes increases, NTP sync time grows significantly, reaching 20 minutes for 11 nodes. Chrony, however, shows a more stable increase in sync time, only reaching 4 minutes for 11 nodes. The graph would highlight the performance gap between NTP and Chrony, with Chrony consistently maintaining lower sync times. For smaller clusters, both protocols perform similarly, but the disparity becomes more pronounced as the cluster size increases. The graph underscores Chrony's efficiency in larger systems.

Cluster Size (Nodes)	NTP Sync Time (minutes)	Chrony Sync Time (minutes)
3	8	1
5	10	1
7	13	2
9	16	2
11	18	3

Table 9: NTP Vs Chrony - 3

Table 9 compares compares the synchronization times for NTP and Chrony across different cluster sizes. As the number of nodes increases, NTP synchronization time grows more significantly, showing that it takes longer to converge and stabilize the system clock in larger clusters. On the other hand, Chrony maintains a much faster synchronization time, even as the cluster size increases. For small clusters with 3 nodes, both NTP and Chrony show relatively low sync times, but Chrony still performs faster. For larger clusters (11 nodes), Chrony takes 3 minutes, while NTP requires 18 minutes. The difference in sync times highlights Chrony's superior performance, especially in dynamic or distributed environments, where fast synchronization is critical. This faster performance is due to Chrony's ability to adapt quickly to network conditions, and its better handling of time drift, unlike NTP, which is slower to adjust. As cluster size increases, the gap between Chrony and NTP becomes more noticeable, underscoring Chrony's scalability and efficiency for larger systems.



Graph 9: NTP Vs Chrony - 3

Graph 9 would show the synchronization times for NTP and Chrony across different cluster sizes. As the number of nodes increases, the NTP sync time increases significantly, while Chrony maintains much faster synchronization times. For small clusters (3 nodes), both NTP and Chrony have low sync times, but Chrony consistently performs better. For larger clusters (11 nodes), Chrony requires only 3 minutes compared to 18 minutes for NTP. The graph visually highlights the increasing disparity in synchronization time as the cluster size grows, emphasizing Chrony's efficiency.

EVALUATION

This evaluation compares the synchronization performance of NTP and Chrony across varying cluster sizes. For a 3-node cluster, NTP sync time is 8 minutes, while Chrony achieves synchronization in just 1 minute. As the cluster size increases to 5, 7, 9, and 11 nodes, NTP sync times rise to 10, 13, 16, and 18 minutes respectively. In contrast, Chrony maintains significantly lower sync times of 1, 2, 2, and 3 minutes across the same configurations. This consistent efficiency demonstrates Chrony's advantage in handling time synchronization in larger distributed systems. While NTP remains reliable, it shows scalability limitations with increased cluster size. Chrony's faster convergence and lower latency make it more suitable for systems where timing precision is critical. The evaluation indicates that for modern distributed applications, especially those requiring rapid synchronization, Chrony may offer more robust performance. Therefore, it is recommended to consider Chrony for deployments where synchronization speed is a priority.

CONCLUSION

The evaluation highlights the performance gap between NTP and Chrony in time synchronization across distributed systems. As cluster size increases, NTP shows a significant rise in sync time, reaching up to 18 minutes for 11 nodes. In contrast, Chrony maintains much lower sync times, staying under 3 minutes even at higher node counts. This demonstrates that Chrony scales more efficiently with minimal delay. NTP, while widely adopted, may introduce latency concerns in large-scale systems. Accurate and fast synchronization is vital for distributed system reliability. Chrony's quicker convergence makes it a preferable choice in such contexts. Systems with real-time or time-sensitive operations benefit from Chrony's performance. Therefore, it is advisable to adopt Chrony in environments where synchronization speed and scalability are critical.

Future Work: Although Chrony is generally simpler than NTP in many scenarios, its setup can still be complex, particularly for users who are not familiar with time synchronization protocols. This is an area that could benefit from further work and improvement in the future.

REFERENCES

[1] Tadayon, T., Time synchronization in distributed systems without a central clock. University of Waterloo. Retrieved from https://uwspace.uwaterloo.ca/items/bce62642-d404-4b7d-bfba-26e8f9c91129, 2019.

[2] Shiral, J., Clock synchronization in distributed systems. Academia.edu. Retrieved from https://www.academia.edu/11074773/Clock_Synchronization_in_Distributed_System, 2014.

[3] Levine, J., Time synchronization over the internet using the network time protocol, IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control, 2002

[4] Decotignie, J. D., The many faces of industrial Ethernet, Proceedings of the IEEE, 2005

[5] Sommer, P., Time synchronization in large-scale distributed systems, Journal of Network and Systems Management, 2009

[6] Mills, D. L., A brief history of NTP time, Computer Communications Review, 2003

[7] Wu, Y. C., Time synchronization in wireless sensor networks, IEEE Signal Processing Magazine, 2011

[8] Kopetz, H., & Ochsenreiter, W., Clock synchronization in distributed real-time systems, IEEE Transactions on Computers, 1987

[9] Ramanathan, P., A survey of techniques for time synchronization in distributed systems, ACM Computing Surveys, 1990

[10] Lenzen, C., Optimal clock synchronization in networks, Distributed Computing Journal, 2010

[11] Ong, P., & Shen, D., Scalable and low-latency distributed systems: Techniques and applications, ACM Computing Surveys, 51(2), 1-30, 2018

[12] De Moura, L., & Bjørner, N., Z3: An efficient SMT solver, Tools and Algorithms for the Construction and Analysis of Systems, 2008, 337-340, 2008

[13] Zhou, X., & Yang, Z., High availability and fault tolerance in distributed systems: A study on the effectiveness of Raft, Springer International Publishing, 2018

[14] McCool, M., & Stefanescu, D., Improving the performance of distributed key-value stores: A case study of etcd, Proceedings of the International Conference on Cloud Computing, 213-222, 2018

[15] Hightower, K., Burns, B., & Beda, J., Kubernetes Up & Running: Dive into the future of cloud-native computing, O'Reilly Media, 2017

[16] Ben-Tovim, A., & Liguori, E., Raft consensus algorithm: A review and comparison with Paxos, Journal of Distributed Computing, 32(6), 307-321, 2017

[17] Wood, R., & Brown, P., The influence of network latency on distributed system performance,

ACM Transactions on Networking, 28(2), 123-136, 2017

[18] Diego, A., & Buda, J., A survey on distributed data stores and consistency models, IEEE Transactions on Cloud Computing, 8(4), 988-1002, 2017

[19] Moser, M., & Gallo, S., Performance analysis of the NTP algorithm for distributed systems, Journal of Computer Science and Technology, 2013

[20] Stevenson, J., & Ahmed, S., Scaling distributed key-value stores for performance and reliability, Journal of Computer Science and Technology, 35(5), 1012-1024, 2017

[21] He, Y., & Zhang, L., Optimizing distributed systems for low-latency reads in large-scale environments, Proceedings of the IEEE/ACM International Symposium on Cloud and Autonomic Computing, 96-103, 2018

[22] Krasnov, P., & May, M., Performance analysis of distributed key-value stores, Proceedings of the International Conference on Distributed Computing Systems, 175-180, 2018