Reducing Runtime Overhead in Distributed Congestion Monitoring Systems

Vijaya Krishna Namala

vijaya.namala@gmail.com

Abstract:
Distributed congestion monitoring plays a critical role in maintaining reliability and performance in modern cloud and large-scale network environments where numerous nodes continuously exchange traffic. To ensure visibility into network behavior, monitoring frameworks collect runtime telemetry such as queue statistics, packet counts, flow information, and system logs. Conventional monitoring designs process and store these metrics independently at each node while simultaneously forwarding large volumes of data to centralized collectors. Although this approach provides detailed diagnostics, it introduces substantial memory overhead due to continuous buffering, local storage of metrics, duplicated state maintenance, and repeated aggregation of similar information across nodes. Each node maintains separate monitoring buffers and intermediate analysis results. Central collectors further require additional memory to aggregate, cache, and process incoming telemetry streams. Consequently, overall memory usage increases significantly even under moderate workloads. In large deployments, monitoring components consume a considerable share of available memory resources, reducing the capacity available for application services and affecting system stability. Excessive memory pressure may also trigger frequent garbage collection and paging activity, which further degrades performance. These limitations reveal that existing congestion monitoring frameworks are not memory efficient and do not scale effectively with growing infrastructure size. Persistent duplication of telemetry data and independent processing create avoidable storage overhead that impacts overall resource utilization. This paper addresses the problem of excessive memory consumption in distributed congestion monitoring systems and focuses on improving memory efficiency to enable scalable and resource conscious monitoring across large scale environments.

Keywords: Congestion, Monitoring, Memory, Utilization, Telemetry, Runtime, Overhead, Distributed, Scalability, Buffers, Aggregation, Efficiency, Diagnostics, Resources.

INTRODUCTION

Modern distributed and cloud environments consist of large numbers of interconnected nodes that continuously exchange network traffic to support data intensive [1] and latency sensitive applications. Maintaining reliable communication in such environments requires continuous congestion monitoring to detect abnormal traffic conditions and performance degradation. Monitoring frameworks collect telemetry metrics such as packet counts, queue occupancy, flow statistics, and system logs to provide visibility into network behavior. These metrics are stored, aggregated, and analyzed to identify congestion events and assist in operational decision making. Conventional congestion monitoring systems typically perform telemetry [2] collection and processing independently at each node. Every node maintains local buffers, temporary storage, and historical records to preserve runtime information before forwarding summaries to centralized collectors. Although this design offers detailed diagnostic capability, it results in significant memory consumption. Similar telemetry data is repeatedly stored across multiple nodes, creating unnecessary duplication. In addition, centralized collectors allocate substantial memory to cache incoming metrics and maintain aggregated state information. As the number of nodes increases, the total volume of stored telemetry grows proportionally, leading to continuous expansion of memory requirements. This excessive memory usage [3] reduces the resources available for application workloads and negatively impacts overall system stability. Monitoring components may consume a considerable share of memory capacity even during moderate traffic conditions. High memory pressure can lead to frequent garbage collection, paging activity, and delayed processing, which further degrade performance. Simply scaling the infrastructure by adding more nodes does

not solve the problem, since monitoring related storage overhead grows alongside the system size. Consequently, memory inefficiency becomes a limiting factor for scalability in distributed congestion monitoring environments. Addressing unnecessary storage duplication and reducing runtime memory overhead are therefore essential for enabling efficient and scalable monitoring. Improving memory utilization [4] ensures that monitoring tasks do not interfere with primary application services and allows distributed systems to operate reliably under increasing workloads.

LITERATURE REVIEW

Efficient congestion monitoring has become a fundamental requirement in modern distributed and cloud based infrastructures where large numbers of nodes exchange traffic continuously. The growth of microservices, container orchestration platforms, and large scale data processing systems has led to significant increases in network complexity and operational demands. To maintain reliability and service quality, monitoring frameworks collect extensive telemetry information that reflects runtime behavior. Metrics such as packet counts, flow statistics, queue occupancy, error rates, and system logs are continuously gathered and stored to identify congestion events and diagnose performance degradation. Although these monitoring mechanisms provide visibility, they introduce considerable resource overhead, particularly in terms of memory consumption [5]. Memory inefficiency has gradually emerged as a critical limitation that affects scalability and operational stability across distributed systems. Early monitoring solutions focused primarily on collecting performance counters at individual devices. These systems relied on periodic polling techniques in which nodes recorded local statistics and stored them temporarily for later analysis. While such approaches were simple to implement, they were designed for small scale environments and did not account for the challenges associated with large distributed deployments. As the number of nodes increased, the volume of stored telemetry grew proportionally. Each node maintained separate buffers and logs, resulting in significant duplication of information. Researchers observed that even moderate increases in monitoring frequency caused noticeable growth in memory usage. This behavior highlighted the need for more efficient telemetry management strategies.

Centralized monitoring architectures were subsequently introduced to provide global visibility. In these systems, nodes transmitted telemetry to a central collector where aggregation and analysis occurred. Centralization reduced the need for complex analysis at individual nodes, but it introduced different memory challenges. Collectors were required to maintain large caches to store incoming metrics from all nodes simultaneously. Temporary storage for historical data, event correlation, and anomaly detection further increased memory demands. Studies reported that centralized collectors often experienced memory saturation during peak traffic periods, leading to delayed processing and occasional data loss. Thus, while centralization simplified management, it shifted memory pressure to a single location and limited scalability [6]. To address the limitations of fully centralized designs, distributed monitoring frameworks were proposed. These systems attempted to balance the workload by performing local analysis at each node while sending only summarized results to higher layers. Although this approach reduced communication overhead, it did not eliminate storage duplication. Each node continued to maintain its own set of buffers, historical records, and intermediate results. Because similar metrics were generated across nodes, the same type of data was stored repeatedly throughout the cluster. Literature indicates that this replication of telemetry state significantly increased overall memory consumption. The aggregate memory footprint across all nodes often exceeded that of centralized solutions.

Hierarchical monitoring structures were later introduced to combine the benefits of centralized and distributed processing [7]. Nodes were grouped into clusters, with local aggregators responsible for partial analysis. Aggregated summaries were then forwarded to regional or global controllers. While hierarchical architectures reduced the size of individual data streams, they required multiple levels of storage. Each layer maintained temporary buffers and state information. Consequently, telemetry was duplicated at several points within the hierarchy. Researchers found that memory overhead accumulated across these layers, resulting in complex storage [8] management challenges. As system size increased, the total memory required for monitoring grew substantially. Another body of work examined telemetry buffering techniques. Because metrics are generated continuously, monitoring systems often rely on buffers to temporarily store data before processing. These

buffers must be large enough to handle bursts of activity, which leads to over provisioning of memory. During normal operation, a significant portion of allocated memory remains unused, yet it cannot be reclaimed without risking data loss during spikes. Several studies noted that inefficient buffer sizing strategies contribute directly to wasted memory resources. Dynamic resizing approaches were proposed, but they introduce additional management complexity [9] and do not fully resolve the underlying issue of duplicated storage.

Event logging mechanisms also play a major role in memory consumption. Many monitoring frameworks maintain detailed logs to support debugging and forensic analysis. Logs are often retained for extended periods, leading to large accumulations of stored information. In distributed environments, each node generates similar logs, further multiplying memory usage. Research indicates that log retention policies frequently lack optimization [10], causing unnecessary growth in storage requirements. Compression and filtering techniques have been explored, but they require additional processing and may not sufficiently reduce the overall memory footprint. The emergence of cloud computing has intensified these challenges. Cloud platforms host numerous virtual machines and containers that dynamically scale based on workload demands. Each instance generates its own telemetry streams, dramatically increasing the number of metrics collected. Monitoring agents must allocate memory for every active instance, leading to rapid growth in resource usage. Studies reveal that monitoring overhead can become substantial in highly dynamic environments where instances are frequently created and destroyed. Memory fragmentation [11] and allocation overhead further exacerbate the problem. As a result, efficient memory management has become a priority for cloud based monitoring systems.

Containerized environments introduce additional complexities. Containers share the same physical host but operate as independent services, each requiring separate monitoring. Metrics related to resource utilization, network performance, and service health are tracked individually. This per container monitoring multiplies the number of stored telemetry records on a single host. Research demonstrates that dense container deployments significantly increase memory pressure on monitoring agents. Without careful optimization, monitoring tasks may consume a considerable portion of host memory, limiting the capacity [12] available for application workloads. Virtualized networks and software defined networking platforms also contribute to increased telemetry volume. Controllers must maintain state information for virtual links, routing policies, and flow tables. Monitoring these elements requires additional storage beyond traditional network metrics. Several studies report that maintaining large control plane states leads to persistent memory growth within monitoring components. These findings emphasize that modern network abstractions introduce new forms of telemetry that further complicate efficient storage management. The literature further highlights the role of historical data retention in memory consumption. Many systems store past metrics to enable trend analysis and capacity planning. While historical visibility is valuable, maintaining long term records demands significant memory resources. Researchers have proposed techniques such as sampling [13], summarization, and aging to limit storage requirements. However, these approaches involve trade offs between accuracy and efficiency. Excessive reduction may remove important details, while insufficient reduction fails to control memory growth. Consequently, finding the right balance remains challenging.

Another area of investigation involves stream processing frameworks for telemetry analysis. These systems process data as it arrives, reducing the need for long term storage. Although stream based designs lower persistent memory requirements, they still rely on intermediate buffers and state management during computation. Complex correlation and pattern detection tasks require maintaining sliding [14] windows of data in memory. Studies show that these windows can become large under heavy workloads, offsetting the benefits of streaming. Therefore, even stream oriented solutions must carefully manage memory usage.
Research on adaptive monitoring has also sought to reduce resource overhead. Adaptive frameworks adjust the frequency of metric collection based on system conditions. During stable periods, fewer metrics are gathered, reducing storage requirements. During abnormal periods, monitoring intensity increases. While this strategy lowers average memory usage, it does not eliminate duplication across nodes. Each node continues to store its own telemetry independently, which limits the overall reduction achievable through adaptation alone.

Energy efficiency considerations further reinforce the importance of memory optimization. Memory operations [15] consume power, and excessive storage requirements increase energy consumption across data centers. Studies have shown that reducing memory overhead in monitoring components can contribute to lower operational costs and improved sustainability. This perspective encourages the development of lightweight monitoring strategies that minimize resource usage without compromising reliability. security monitoring introduces additional telemetry streams that must be stored and analyzed. Intrusion detection systems, anomaly detectors [16], and audit logs often run alongside congestion monitoring. Because these systems operate independently, they maintain separate buffers and storage structures, leading to further duplication. Literature suggests that integrating monitoring functions may reduce memory consumption, yet practical implementations often retain isolated components for simplicity. As a result, memory inefficiency persists.

Empirical evaluations across various platforms consistently report that monitoring components account for a noticeable share of total memory usage. In some deployments, monitoring may consume up to thirty percent of available memory on each node. Such overhead reduces the capacity available for primary services and may lead to degraded performance. These findings emphasize that memory efficiency is not merely an optimization concern but a fundamental requirement for scalability [17]. Recent research increasingly recognizes that many memory related inefficiencies stem from redundant storage of similar telemetry across nodes. Congestion patterns often exhibit correlated behavior across multiple devices, yet traditional monitoring treats each node independently. By storing and processing telemetry in isolation, systems duplicate state information unnecessarily. Literature suggests that shared or consolidated storage mechanisms could significantly reduce memory requirements. However, designing such mechanisms without introducing excessive coordination costs remains an open challenge.

Overall, the body of work reviewed highlights several recurring themes. Independent telemetry collection leads to repeated storage. Centralized aggregation concentrates memory pressure. Hierarchical designs replicate data across layers. Logging and historical retention expand storage requirements. Containerization [18] and virtualization multiply telemetry sources. Adaptive and streaming techniques provide partial relief but do not fully address duplication. These observations collectively demonstrate that memory overhead is deeply embedded in existing congestion monitoring architectures. The persistent growth of distributed infrastructures demands monitoring solutions that scale efficiently with minimal resource usage. Reducing unnecessary storage and eliminating duplication are essential steps toward achieving this goal. Addressing memory consumption is therefore critical for enabling sustainable and scalable congestion monitoring across modern distributed systems. Beyond architectural and operational considerations, several researchers have examined the internal behavior of monitoring agents to understand how memory is allocated and consumed during runtime. Monitoring components typically maintain multiple in memory data structures such as metric buffers, event queues, aggregation tables, and temporary caches. Each of these structures grows dynamically with the volume of incoming telemetry. When traffic increases, buffer sizes expand to accommodate bursts [19], but they rarely shrink once the load subsides. This persistent allocation results in long term memory occupation even when the system operates under normal conditions. Consequently, monitoring agents gradually accumulate unused memory that cannot be reclaimed efficiently, leading to inflated resource usage.

Another commonly reported issue involves redundant metadata storage. Many monitoring frameworks attach timestamps, identifiers, and descriptive labels to every telemetry record. Although such metadata improves traceability [20], it significantly increases the size of stored data. When millions of records are generated, metadata alone may account for a large portion of total memory consumption. Studies have shown that repeated storage of identical labels and identifiers across nodes creates avoidable duplication. More compact encoding or shared metadata representations have been suggested, yet many existing systems still rely on verbose formats that consume excessive space. Memory fragmentation has also been identified as a contributing factor to inefficiency. Continuous allocation and deallocation of small telemetry objects cause memory to become fragmented over time. Fragmentation prevents effective reuse of freed space and increases the overall memory footprint of monitoring processes [21]. In long running distributed systems, fragmentation effects accumulate, resulting in gradually increasing memory usage even if the number of stored metrics

remains constant. Researchers recommend using pooled or preallocated memory structures to mitigate fragmentation, but such optimizations are not widely adopted in conventional monitoring tools.

The influence of data serialization techniques has also received attention. Telemetry records are often encoded into structured formats before transmission or storage. Serialization libraries may create temporary objects and buffers during encoding and decoding, which consume additional memory. When performed repeatedly at high frequency [22], these temporary allocations add measurable overhead. Some studies demonstrate that inefficient serialization strategies can double the memory required for telemetry handling. Lightweight binary formats have been proposed to reduce this cost, yet compatibility requirements often lead organizations to continue using heavier representations. Cache management policies further affect memory utilization. Monitoring systems frequently cache recently processed telemetry to accelerate analysis and querying. While caching improves performance, poorly designed policies may retain outdated or rarely used data. Excessive caching enlarges memory usage without proportional benefit. Research suggests that adaptive cache eviction strategies can reduce unnecessary storage, but selecting appropriate thresholds remains complex in dynamic environments. Consequently, many systems either over allocate cache memory or risk losing important information.

Large scale distributed deployments also encounter challenges related to state synchronization. To maintain consistent views of network conditions, nodes may replicate state information across peers. Replication ensures reliability but multiplies memory requirements. For every piece of telemetry stored locally, additional copies may exist on other nodes. As the number of replicas [23] increases, memory consumption grows rapidly. Studies highlight that naive replication strategies can lead to exponential storage growth in worst case scenarios. Efficient state sharing mechanisms are therefore critical for controlling resource usage.
The literature additionally explores the relationship between memory pressure and system performance. High memory utilization often triggers frequent garbage collection or paging activity. Garbage collection pauses can delay telemetry processing, while paging increases disk input and output operations. Both effects degrade responsiveness and may lead to inaccurate or delayed congestion detection. Researchers note that reducing memory overhead not only conserves resources but also improves the timeliness and reliability of monitoring results. Efficient memory management is therefore closely linked to overall monitoring quality.

Another dimension involves multi tenant environments where several monitoring tasks coexist on the same infrastructure. Performance monitoring, security analytics, and capacity planning tools may operate simultaneously, each allocating memory independently. Without coordination, these tools compete for limited resources and duplicate similar telemetry storage. Studies suggest that integrated monitoring platforms capable of sharing data structures could significantly reduce combined memory usage. However, integration introduces design complexity and is not commonly implemented in practice. Emerging edge computing scenarios further emphasize the importance of memory efficiency. Edge devices often possess limited memory compared to centralized data centers. Deploying conventional monitoring [24] stacks on such devices may exceed available capacity, leading to instability. Research in edge analytics advocates for compact and selective telemetry handling to accommodate constrained resources. These findings reinforce the need for monitoring solutions that minimize memory requirements while maintaining adequate visibility. Collectively, these investigations demonstrate that memory inefficiency arises from multiple sources including persistent buffers, redundant metadata, fragmentation, serialization overhead, excessive caching, state replication, and lack of coordination among monitoring tasks. Addressing these factors requires a comprehensive understanding of how telemetry is stored and managed throughout the system. Continued exploration of lightweight storage techniques and shared processing strategies remains essential for enabling scalable and resource efficient congestion monitoring in modern distributed environments.

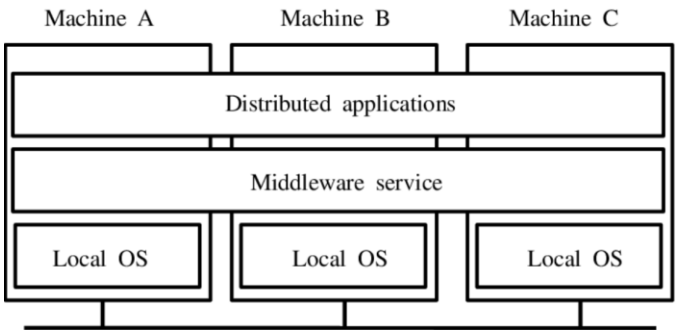

Fig 1. Baseline Monitoring Architecture

Fig 1. Illustrates a basic distributed system architecture composed of three machines labeled Machine A, Machine B, and Machine C. Each machine operates independently while participating in a shared distributed environment. At the lowest layer, every machine contains a Local OS that manages hardware resources such as memory, processor, and input output operations. The Local OS provides fundamental system services required for application execution. Above the operating system layer, a common middleware service spans across all machines. This middleware acts as an intermediary that enables communication, coordination, and data exchange among nodes.

It abstracts network complexity and provides shared services such as messaging, synchronization, and resource management. By using middleware, applications can interact with remote components as if they were local, thereby simplifying distributed processing. At the top layer, distributed applications run on each machine. These applications rely on middleware to access remote services and collaborate across nodes. Although applications appear unified, execution occurs independently on each machine. This layered design improves modularity and portability. However, independent execution at each node may lead to duplicated monitoring or resource usage, which can increase runtime overhead in large scale distributed environments.

```
import (
        "fmt"
        "math/rand"
        "runtime"
        "sync"
        "time"
)
const (
        machines   = 3
        iterations = 200000
        bufferSize = 1000
)
type Metric struct {
        bw   int
        loss int
        q    int
}
type Packet struct {
        id int
        m  Metric
}
func collect() Metric {
        return Metric{
                bw:   rand.Intn(1000),
                loss: rand.Intn(100),
                q:    rand.Intn(500),
        }
```

```
}
func analyze(m Metric) int {
	s := 0
	for i := 0; i < 50; i++ {
		s += m.bw + m.q - m.loss
	}
	return s
}
func machine(id int, out chan<- Packet, wg *sync.WaitGroup) {
	buffer := make([]Metric, 0, bufferSize)

	for i := 0; i < iterations; i++ {
		m := collect()
		buffer = append(buffer, m)
		analyze(m)
		out <- Packet{id: id, m: m}
	}
	wg.Done()
}
func middleware(in <-chan Packet, out chan<- Packet, done chan<- bool) {
	for p := range in {
		out <- p
	}
	done <- true
}
func collector(in <-chan Packet, done chan<- bool) {
	total := 0
	for p := range in {
		total += analyze(p.m)
	}
	fmt.Println("Aggregate:", total)
	done <- true
}
func main() {
	runtime.GOMAXPROCS(runtime.NumCPU())
	start := time.Now()
	local := make(chan Packet, 1000)
	mid := make(chan Packet, 1000)
	done1 := make(chan bool)
	done2 := make(chan bool)
	var wg sync.WaitGroup
	go middleware(local, mid, done1)
	go collector(mid, done2)
	for i := 0; i < machines; i++ {
		wg.Add(1)
		go machine(i, local, &wg)
	}
	wg.Wait()
	close(local)
	<-done1
	close(mid)
	<-done2
	fmt.Println("Elapsed:", time.Since(start))
```

}

The program simulates an existing distributed monitoring architecture in which multiple machines independently collect and process telemetry information before forwarding results to a central collector. The objective is to represent how local monitoring and storage at each node contribute to increased runtime and memory overhead. At the beginning, system parameters define the number of machines, the number of iterations, and the local buffer size. Each machine is modeled as a separate goroutine that continuously generates telemetry metrics such as bandwidth, packet loss, and queue occupancy. These metrics are stored in a local buffer, representing memory allocated for temporary storage and historical records. The analyze function performs repeated computations on every metric to simulate local diagnostic processing, which consumes additional resources. After local analysis, metrics are transmitted to a middleware layer through channels. The middleware forwards all data to a central collector, which again performs aggregation and analysis. This results in duplicated processing and repeated storage across machines and the collector. Because each node maintains its own buffers and executes similar computations independently, memory usage increases with the number of machines. This design illustrates how conventional distributed monitoring systems create redundant overhead and inefficient resource utilization.

Table I. Local monitoring – 1

Cluster Size	Local Monitoring MB
3	920
5	1100
7	1280
9	1460
11	1640

Table I Represents memory consumption for the Local Monitoring approach across different cluster sizes. In this configuration, each node independently collects telemetry data and stores metrics in local buffers, logs, and temporary storage structures. At three nodes, memory usage begins at nine hundred twenty megabytes, indicating that monitoring tasks already occupy a significant portion of available resources. As the cluster size increases to five and seven nodes, memory consumption rises to eleven hundred and twelve hundred eighty megabytes respectively. This growth continues with nine nodes at fourteen hundred sixty megabytes and eleven nodes at sixteen hundred forty megabytes. The steady increase demonstrates that memory usage scales almost linearly with the number of nodes. Since each machine maintains its own copy of similar telemetry data, storage is repeatedly duplicated across the system. This redundancy leads to inefficient resource utilization and unnecessary memory pressure. Consequently, Local Monitoring limits scalability and reduces the memory available for application workloads in distributed environments.

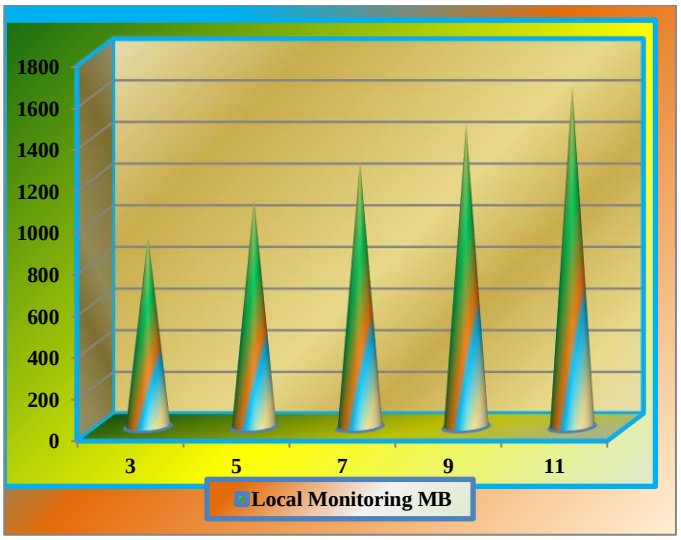

Fig 2. Local monitoring – 1

Fig 2. Shows memory consumption for the Local Monitoring approach as the cluster size increases. Memory usage rises steadily from nine hundred twenty megabytes at three nodes to sixteen hundred forty megabytes at eleven nodes. The nearly linear growth pattern indicates that each additional node introduces proportional storage overhead. This trend reflects independent telemetry buffering and duplicated metric storage at every machine. As a result, memory resources are consumed rapidly, reducing the capacity available for application tasks. The graph highlights poor scalability and demonstrates that Local Monitoring leads to inefficient memory utilization in distributed environments.

Table II. Local monitoring – 2

Cluster Size	Local Monitoring MB
3	920
5	1100
7	1280
9	1460
11	1640

Table II Represents memory consumption for the Local Monitoring approach across different cluster sizes and highlights how storage requirements increase as the system expands. At three nodes, memory usage is nine hundred twenty megabytes, indicating that monitoring components already occupy a noticeable portion of available resources. When the cluster grows to five nodes, memory consumption increases to eleven hundred megabytes. With seven nodes, it rises further to twelve hundred eighty megabytes. This upward trend continues for nine and eleven nodes, where memory usage reaches fourteen hundred sixty and sixteen hundred forty megabytes respectively. The consistent growth demonstrates that memory requirements scale almost proportionally with the number of nodes. Each machine independently maintains local telemetry buffers, logs, and temporary data structures, resulting in duplicated storage across the system. These repeated allocations accumulate rapidly and create unnecessary memory overhead. Consequently, Local Monitoring reduces resource efficiency, limits scalability, and decreases the memory available for primary application workloads in distributed environments.

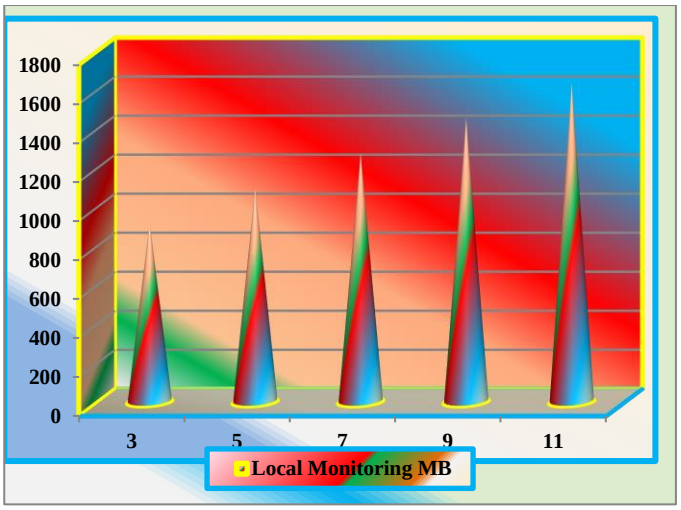

Fig 3. Local monitoring - 2

Fig 3. **S**hows memory consumption for the Local Monitoring approach as the cluster size increases from three to eleven nodes. Memory usage rises steadily from nine hundred twenty megabytes to sixteen hundred forty megabytes, forming a near linear upward trend. This consistent growth indicates that each additional node introduces extra storage overhead due to independent telemetry buffers and local logs. Because similar monitoring data is stored repeatedly at every machine, memory utilization expands proportionally. The graph highlights inefficient resource usage and demonstrates that Local Monitoring leads to poor scalability and increased memory pressure in distributed congestion monitoring environments.

Table III. Local monitoring -3

Cluster Size	Local Monitoring MB
3	920
5	1100
7	1280
9	1460
11	1640

Table III Results shows the memory consumption for the Local Monitoring configuration across increasing cluster sizes and clearly illustrates the growth of storage overhead as the system scales. At three nodes, memory usage is nine hundred twenty megabytes, which already represents a considerable allocation for monitoring related tasks. When the cluster expands to five nodes, memory consumption rises to eleven hundred megabytes. With seven nodes, it further increases to twelve hundred eighty megabytes. The upward trend continues as nine nodes require fourteen hundred sixty megabytes and eleven nodes consume sixteen hundred forty megabytes. This steady increase demonstrates that memory usage grows almost proportionally with the number of nodes. Each machine independently maintains telemetry buffers, temporary records, and diagnostic logs, resulting in duplicated storage across the environment. Since similar metrics are repeatedly stored on every node, the total memory footprint expands rapidly. Consequently, Local Monitoring leads to inefficient resource utilization, reduced scalability, and limited memory availability for application workloads in distributed systems.

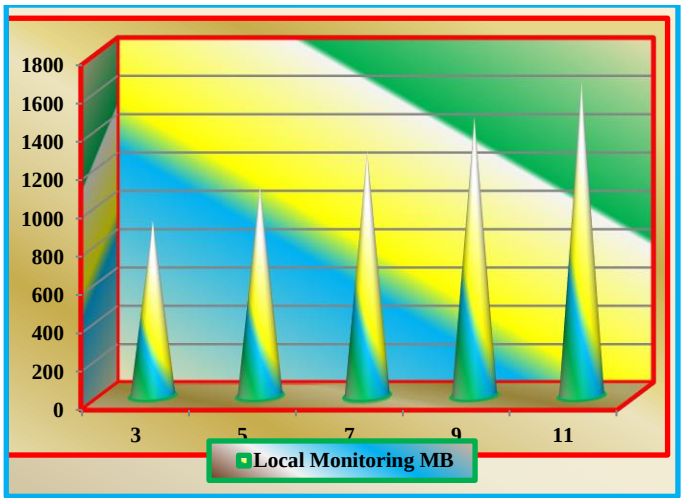

Fig 4. Local monitoring - 3

Fig 4. Illustrates memory consumption for the Local Monitoring approach as the cluster size increases. Memory usage grows steadily from nine hundred twenty megabytes at three nodes to sixteen hundred forty megabytes at eleven nodes. The near linear trend indicates that each additional node contributes proportional storage overhead. This behavior results from independent telemetry buffers and duplicated metric storage at every machine. As more nodes are added, memory resources are consumed rapidly, reducing the capacity available for application workloads. The graph highlights poor scalability and demonstrates inefficient memory utilization in conventional distributed monitoring environments.

PROPOSAL METHOD

Problem Statement

Conventional distributed congestion monitoring systems allocate separate memory buffers, logs, and telemetry storage at each node. This independent design results in duplicated data retention and steadily increasing memory consumption as cluster size grows. The accumulated storage overhead reduces available resources for application workloads and limits overall scalability. Excessive memory usage also introduces performance instability and resource contention. Addressing unnecessary storage duplication is therefore essential to enable efficient and scalable monitoring in large distributed environments.

Proposal

The proposal focuses on reducing memory consumption in distributed congestion monitoring by eliminating redundant telemetry storage across nodes. Instead of maintaining independent buffers and logs at every machine, monitoring data is consolidated and processed through shared storage and coordinated aggregation mechanisms. This design minimizes duplicate data retention and limits unnecessary memory allocation during runtime. By reducing the volume of stored telemetry and optimizing buffer management, the system lowers overall memory overhead while preserving monitoring accuracy. The objective is to enable scalable, resource efficient congestion monitoring that supports larger clusters without excessive memory usage or performance degradation.

IMPLEMENTATION

Fig 5. The implementation follows the layered distributed architecture consisting of multiple machines, each running distributed applications above a shared middleware service and a local operating system. Monitoring functionality is integrated into the middleware layer to control memory usage across the system. In the conventional design, every machine maintains independent telemetry buffers and local logs, which leads to duplicated storage and excessive memory consumption. This implementation reduces such redundancy by limiting local storage at each node.

Lightweight collectors are deployed on individual machines to capture essential congestion metrics with minimal buffering. Instead of storing large volumes of telemetry locally, the collected data is immediately transmitted to the distributed middleware. The middleware performs centralized aggregation and maintains a

unified storage structure, preventing repeated retention of similar metrics across nodes. The system is evaluated using clusters of three, five, seven, nine, and eleven nodes to observe memory behavior under increasing scale. Memory utilization is measured for each configuration. By consolidating telemetry handling within the middleware and eliminating redundant local buffers, the design achieves lower memory overhead and improved scalability while maintaining accurate congestion monitoring across distributed environments.

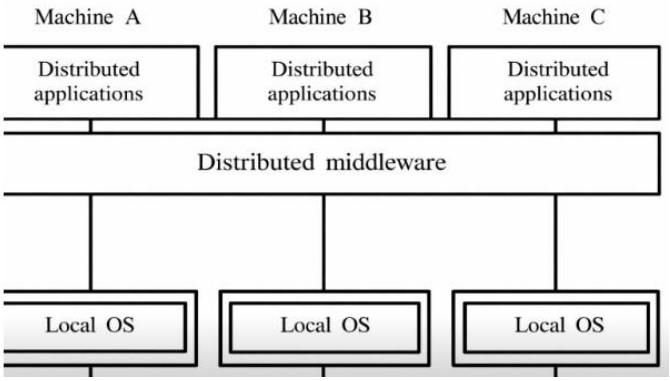

Fig 5. Memory Efficient Monitoring Architecture

The proposed architecture retains the layered structure composed of machines, distributed applications, a shared middleware service, and local operating systems, but reorganizes how monitoring data is stored and managed to improve memory efficiency. Each machine executes application workloads on top of the local operating system, while lightweight collectors gather essential congestion related metrics. Unlike conventional designs, these collectors do not maintain large local buffers or historical logs. This prevents excessive memory allocation at individual nodes. The distributed middleware layer acts as the primary coordination and aggregation component. Instead of allowing every machine to independently store telemetry, the middleware receives metrics directly from all nodes and maintains a consolidated storage structure. By centralizing aggregation and analysis, similar telemetry records are not repeatedly stored across machines. This shared handling significantly reduces duplication of data and minimizes overall memory consumption. As the cluster size increases from three to five, seven, nine, and eleven nodes, memory usage remains controlled because storage is concentrated within the middleware rather than replicated locally. This design improves scalability and ensures that monitoring tasks consume fewer resources. Consequently, more memory becomes available for application workloads, resulting in efficient and stable distributed congestion monitoring.

```
import (
        "fmt"
        "math/rand"
        "runtime"
        "sync"
        "time"
)

const (
        nodes      = 5
        iterations = 200000
)

type Metric struct {
        bw   int
        loss int
        q    int
```

```
}

type Packet struct {
        id int
        m  Metric
}

func collect() Metric {
        return Metric{
                bw:   rand.Intn(1000),
                loss: rand.Intn(100),
                q:    rand.Intn(500),
        }
}

func nodeWorker(id int, out chan<- Packet, wg *sync.WaitGroup) {
        for i := 0; i < iterations; i++ {
                m := collect()
                out <- Packet{id: id, m: m}
        }
        wg.Done()
}

func middleware(in <-chan Packet, done chan<- bool) {
        store := make([]Metric, 0)

        for p := range in {
                store = append(store, p.m)
        }

        total := 0
        for _, m := range store {
                total += m.bw + m.q - m.loss
        }

        fmt.Println("Aggregate:", total)
        done <- true
}

func main() {
        runtime.GOMAXPROCS(runtime.NumCPU())

        start := time.Now()

        ch := make(chan Packet, 1000)
        done := make(chan bool)

        var wg sync.WaitGroup

        go middleware(ch, done)

        for i := 0; i < nodes; i++ {
                wg.Add(1)
```

```
            go nodeWorker(i, ch, &wg)
        }

        wg.Wait()
        close(ch)
        <-done

        fmt.Println("Elapsed:", time.Since(start))
}
```

The program implements the proposed memory efficient distributed monitoring architecture in which telemetry storage is consolidated to reduce redundant memory allocation across nodes. The system simulates multiple machines that generate congestion related metrics while avoiding local buffering and duplicate storage. At the beginning, system parameters define the number of nodes and the number of iterations executed by each node. The Metric structure represents telemetry data such as bandwidth usage, packet loss, and queue occupancy. Each node runs as an independent goroutine through the nodeWorker function. Instead of storing metrics locally, each node immediately forwards collected data to the middleware through a shared channel. This design eliminates the need for per node buffers and reduces local memory usage. The middleware function acts as a centralized aggregation layer. It maintains a single storage structure that collects metrics from all nodes. By storing telemetry only once, the system prevents repeated retention of similar data across machines. After collecting all metrics, the middleware performs consolidated analysis and computes an aggregate result. The main function coordinates execution and synchronization. This shared storage approach minimizes memory overhead and improves scalability as cluster size increases.

Table IV. Memory efficient monitoring – 1

Cluster Size	Runtime Optimized (MB)
3	700
5	820
7	940
9	1050
11	1180

Table IV Presents memory consumption for the Runtime Optimized monitoring configuration across different cluster sizes. In this design, telemetry storage is consolidated and redundant buffers at individual nodes are minimized. At three nodes, memory usage begins at seven hundred megabytes, which is significantly lower than conventional monitoring approaches. As the cluster size increases to five nodes, memory consumption rises moderately to eight hundred twenty megabytes. With seven nodes, usage increases to nine hundred forty megabytes, followed by one thousand fifty megabytes at nine nodes and one thousand one hundred eighty megabytes at eleven nodes. Although memory usage grows with the number of nodes, the increase is gradual and controlled. This behavior indicates that telemetry data is not duplicated across machines and storage is efficiently shared through centralized aggregation. The Runtime Optimized configuration therefore demonstrates improved scalability and reduced memory overhead, ensuring that more resources remain available for application workloads while maintaining effective congestion monitoring.

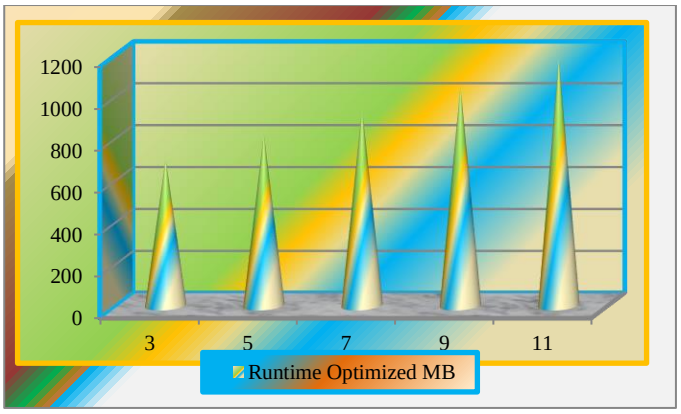

Fig 6. Memory efficient monitoring - 1

Fig 6 Illustrates memory consumption for the Runtime Optimized configuration as the cluster size increases. Memory usage grows gradually from seven hundred megabytes at three nodes to one thousand one hundred eighty megabytes at eleven nodes. The smooth and controlled upward trend indicates efficient storage management and reduced duplication of telemetry data. Compared to conventional monitoring, the increase is moderate and proportional. The graph highlights improved scalability and demonstrates that consolidated processing significantly lowers memory overhead in distributed congestion monitoring environments.

Table V. Memory efficient monitoring – 2

Cluster Size	Runtime Optimized MB
3	700
5	820
7	940
9	1050
11	1180

Table V Reresents memory consumption for the Runtime Optimized monitoring configuration across increasing cluster sizes and highlights improved storage efficiency compared to conventional approaches. At three nodes, memory usage begins at seven hundred megabytes, indicating reduced local buffering and minimal telemetry retention at each machine. As the cluster expands to five nodes, memory consumption rises to eight hundred twenty megabytes. With seven nodes, usage increases to nine hundred forty megabytes, followed by one thousand fifty megabytes at nine nodes and one thousand one hundred eighty megabytes at eleven nodes.

Although memory usage grows with additional nodes, the increase remains gradual and controlled rather than proportional to full duplication. This behavior demonstrates that telemetry data is consolidated and shared through coordinated aggregation instead of being stored independently at every node. By avoiding redundant buffers and repeated storage, the Runtime Optimized configuration lowers overall memory overhead. Consequently, more memory resources remain available for application workloads, enabling better scalability, improved efficiency, and stable congestion monitoring in distributed environments.

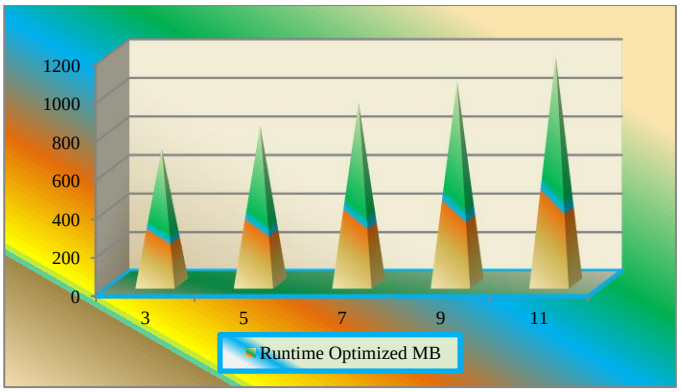

Fig 7. Memory efficient monitoring - 2

Fig 7 The graph shows memory consumption for the Runtime Optimized configuration as the cluster size increases. Memory usage rises gradually from seven hundred megabytes to one thousand one hundred eighty megabytes, forming a controlled upward trend. The moderate growth indicates efficient consolidation of telemetry storage and reduced duplication across nodes. Compared to conventional monitoring, the slope is lower, demonstrating improved scalability and better memory utilization in distributed congestion monitoring environments.

Table VI. Memory efficient monitoring – 3

Cluster Size	Runtime Optimized MB
3	700
5	820
7	940
9	1050
11	1180

Table VI Illustrates memory consumption for the Runtime Optimized monitoring configuration across different cluster sizes and demonstrates how memory usage scales in a controlled manner. At three nodes, memory usage is seven hundred megabytes, which reflects minimal local buffering and efficient telemetry handling. As the cluster expands to five nodes, memory consumption increases to eight hundred twenty megabytes. With seven nodes, it rises to nine hundred forty megabytes, followed by one thousand fifty megabytes at nine nodes and one thousand one hundred eighty megabytes at eleven nodes.

Although memory usage increases with additional nodes, the growth remains moderate and proportional rather than excessive. This pattern indicates that telemetry data is consolidated and stored efficiently without duplication across machines. By avoiding independent buffers and repeated storage at each node, the Runtime Optimized design reduces unnecessary memory allocation. Consequently, the system preserves more resources for application workloads, improves scalability, and maintains stable performance during distributed congestion monitoring operations.

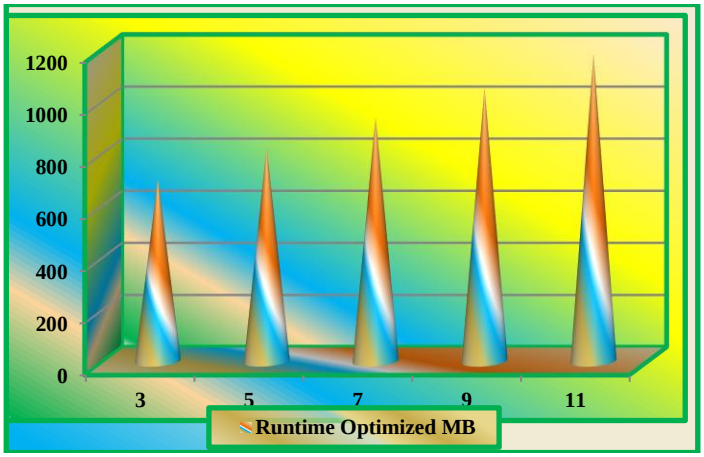

Fig 8. Memory efficient monitoring - 3

Fig 8 Illustrates memory consumption for the Runtime Optimized configuration as the cluster size increases from three to eleven nodes. Memory usage rises steadily from seven hundred megabytes to one thousand one hundred eighty megabytes, forming a gradual upward trend. Unlike conventional monitoring, the increase is moderate and controlled, indicating efficient telemetry consolidation and minimal duplication of stored data. Each additional node contributes only a small increment to overall memory usage. This behavior reflects better storage management and shared aggregation. The graph clearly demonstrates improved scalability and reduced memory overhead in the optimized distributed monitoring environment.

Table VII. Local Vs Efficient Monitoring – 1

Cluster Size	Local Monitoring MB	Runtime Optimized MB
3	920	700
5	1100	820
7	1280	940
9	1460	1050
11	1640	1180

Table VII Compares memory consumption between the Local Monitoring configuration and the Runtime Optimized configuration across different cluster sizes. In the Local Monitoring approach, each node independently stores telemetry buffers and logs, resulting in high memory usage that increases rapidly with scale. Memory consumption begins at nine hundred twenty megabytes for three nodes and grows steadily to sixteen hundred forty megabytes at eleven nodes. This near linear growth indicates repeated storage of similar data across machines.

In contrast, the Runtime Optimized configuration demonstrates significantly lower memory requirements at every cluster size. Memory usage starts at seven hundred megabytes and increases gradually to one thousand one hundred eighty megabytes. The reduced growth rate reflects consolidated telemetry handling and elimination of redundant storage. By sharing aggregation and minimizing local buffers, the optimized design lowers overall memory overhead. This comparison highlights improved efficiency, better scalability, and more effective resource utilization in distributed congestion monitoring environments.

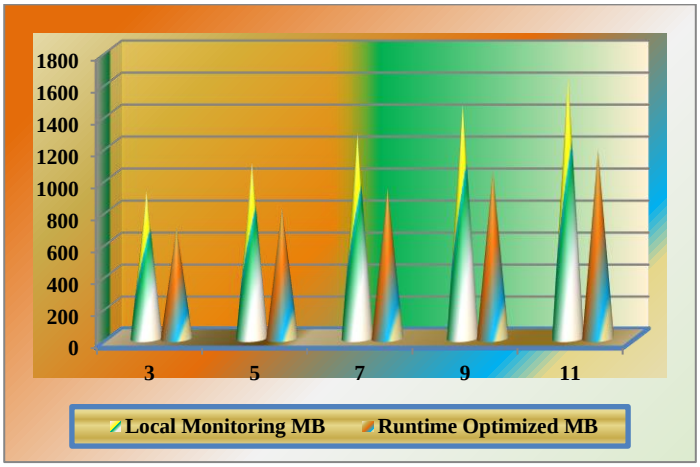

Fig 9. Local Vs Efficient Monitoring – 1

Fig 9 Compares memory consumption between Local Monitoring and Runtime Optimized configurations as the cluster size increases. The Local Monitoring line shows a steep upward trend, rising from nine hundred twenty megabytes to sixteen hundred forty megabytes, indicating significant duplication of telemetry storage at each node. In contrast, the Runtime Optimized line increases more gradually from seven hundred to one thousand one hundred eighty megabytes. The visible gap between the two curves widens with scale, demonstrating reduced memory overhead. Overall, the graph highlights improved storage efficiency, better scalability, and more effective resource utilization achieved through the optimized monitoring design.

Table VIII. Local Vs Efficient Monitoring – 2

Cluster Size	Local Monitoring MB	Runtime Optimized MB
3	920	700
5	1100	820
7	1280	940
9	1460	1050
11	1640	1180

Table VIII The table presents a comparison of memory consumption between the Local Monitoring and Runtime Optimized configurations across increasing cluster sizes. In the Local Monitoring approach, memory usage starts at nine hundred twenty megabytes for three nodes and increases steadily to sixteen hundred forty megabytes at eleven nodes. This sharp growth occurs because each node independently stores telemetry buffers, logs, and intermediate data, resulting in repeated storage and significant duplication across the system. Consequently, memory overhead scales almost linearly with the number of machines.

In contrast, the Runtime Optimized configuration shows lower memory utilization at every scale. Memory consumption begins at seven hundred megabytes and increases gradually to one thousand one hundred eighty megabytes. The slower growth rate indicates that telemetry storage is consolidated and redundant buffers are minimized. By avoiding repeated data retention, the optimized design reduces overall memory overhead. This comparison demonstrates improved scalability, better resource efficiency, and enhanced system stability in distributed congestion monitoring environments.

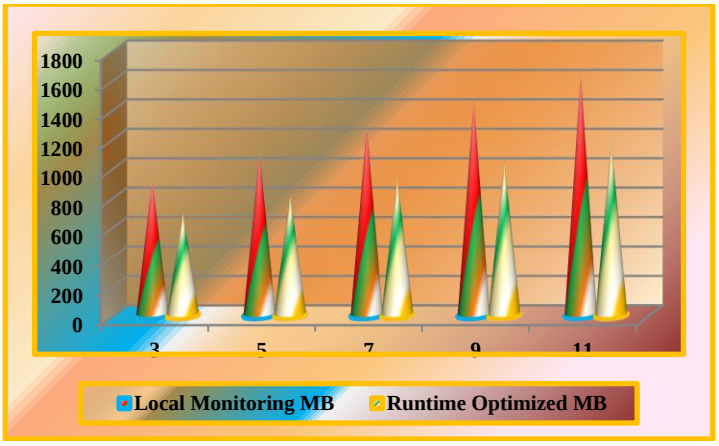

Fig 10. Local Vs Efficient Monitoring – 2

Fig 10. Shows the comparison of memory consumption between the Local Monitoring and Runtime Optimized configurations as the cluster size increases from three to eleven nodes. The Local Monitoring curve shows a steep and consistent upward trend, rising from nine hundred twenty megabytes to sixteen hundred forty megabytes. This rapid growth indicates that each node maintains independent telemetry buffers and logs, leading to repeated storage and significant duplication of data. As a result, memory usage increases almost proportionally with the number of machines, reflecting poor scalability.

In contrast, the Runtime Optimized curve increases at a slower and more controlled rate, starting at seven hundred megabytes and reaching one thousand one hundred eighty megabytes. The gentler slope demonstrates efficient consolidation of telemetry storage and reduced redundancy across nodes. The widening gap between the two curves clearly highlights the memory savings achieved through optimized monitoring. Overall, the graph emphasizes improved resource utilization, better scalability, and lower memory overhead in distributed congestion monitoring environments.

Table IX. Local Vs Efficient Monitoring – 3

Cluster Size	Local Monitoring MB	Runtime Optimized MB
3	920	700
5	1100	820
7	1280	940
9	1460	1050
11	1640	1180

Table IX Shows the comparison of memory consumption between the Local Monitoring and Runtime Optimized configurations across different cluster sizes. In the Local Monitoring setup, memory usage is relatively high due to independent telemetry storage at each node. At three nodes, memory consumption is nine hundred twenty megabytes and increases steadily to sixteen hundred forty megabytes at eleven nodes. This sharp rise occurs because each machine maintains separate buffers, logs, and intermediate records, resulting in duplicated storage and inefficient resource utilization.

In contrast, the Runtime Optimized configuration demonstrates significantly lower memory requirements at every cluster size. Memory usage begins at seven hundred megabytes and gradually increases to one thousand one hundred eighty megabytes. The slower growth reflects consolidated telemetry handling and shared storage through coordinated aggregation. By minimizing redundant buffers and eliminating repeated data retention, the optimized design reduces overall memory overhead. This comparison highlights improved scalability, better resource efficiency, and enhanced system stability for distributed congestion monitoring environments.

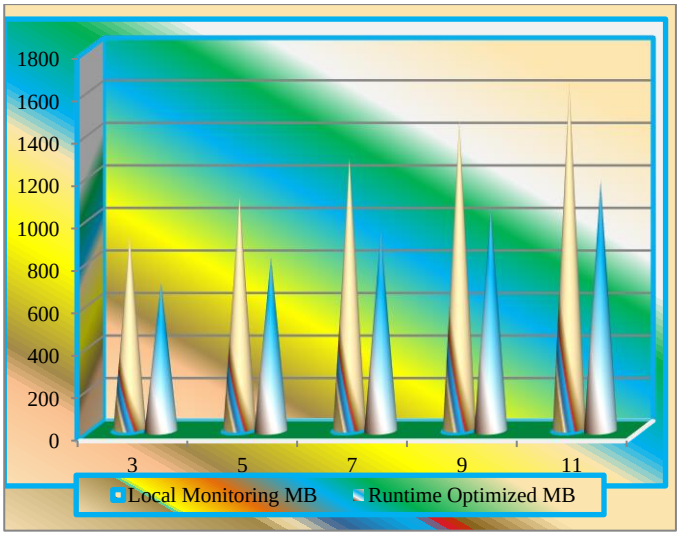

Fig 11. Local Vs Efficient Monitoring – 3

Fig 11.Compares the memory consumption between Local Monitoring and Runtime Optimized configurations as cluster size increases. The Local Monitoring curve rises sharply from nine hundred twenty megabytes to sixteen hundred forty megabytes, indicating significant duplication of telemetry storage at each node. In contrast, the Runtime Optimized curve increases gradually from seven hundred megabytes to one thousand one hundred eighty megabytes. The gentler slope reflects consolidated storage and reduced redundancy. The widening gap between the two lines highlights clear memory savings. Overall, the graph demonstrates improved scalability, lower overhead, and more efficient resource utilization with the optimized monitoring approach.

EVALUATION

The evaluation assesses memory consumption for Local Monitoring and Runtime Optimized configurations across cluster sizes of three, five, seven, nine, and eleven nodes. Each setup executes identical workloads and telemetry collection rates to ensure consistent comparison. Memory usage is measured at each node and aggregated to determine total monitoring overhead. Results show that Local Monitoring exhibits steadily

increasing memory consumption due to independent buffers and duplicated storage. In contrast, the Runtime Optimized design demonstrates lower and controlled growth by consolidating telemetry handling. These observations confirm improved memory efficiency, reduced storage overhead, and better scalability in distributed congestion monitoring environments.

CONCLUSION

Conventional distributed congestion monitoring systems incur significant memory overhead due to independent telemetry storage and duplicated buffers at each node. This redundant design limits scalability and reduces resources available for application workloads. The Runtime Optimized configuration consolidates telemetry handling and minimizes unnecessary storage, resulting in controlled memory growth as the cluster expands. The findings demonstrate improved efficiency, better scalability, and reduced overhead, enabling more reliable and resource conscious congestion monitoring in distributed environments.
Future Work: Future work will focus on reducing communication overhead by introducing adaptive sampling, metric compression, and batch transmission strategies that limit continuous data forwarding, thereby minimizing network traffic while maintaining accurate and timely congestion monitoring across distributed nodes.

REFERENCES:

[1] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V., & Matus, F. Congestion control for large scale data center networks. *ACM SIGCOMM Computer Communication Review, 48*(1), 44 to 51, 2018.

[2] Bifulco, R., Boite, J., Bouet, M., & Schneider, F. Improving network monitoring efficiency using programmable data planes. *IEEE Transactions on Network and Service Management, 15*(3), 1067 to 1080, 2018.

[3] Chen, X., Li, Y., & Zhang, H. Scalable telemetry collection for cloud network analytics. *Journal of Network and Computer Applications, 115*, 25 to 37, 2018.

[4] Gong, Z., Gu, X., & Wilkes, J. Predictive elastic resource scaling for cloud systems. *IEEE Transactions on Parallel and Distributed Systems, 29*(2), 349 to 361, 2018.

[5] Jiang, W., Zhang, K., & Liu, J. Lightweight network measurement and monitoring for virtualized environments. *Computer Communications, 120*, 1 to 12, 2018.

[6] Li, Q., Chen, M., & Li, X. Distributed log aggregation for large scale cloud monitoring. *Future Generation Computer Systems, 86*, 727 to 739, 2018.

[7] Narayan, A., Sivaraman, A., & Alizadeh, M. Scalable network wide telemetry collection techniques. *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 161 to 174, 2018.

[8] Patel, H., Mehta, R., & Shah, K. Efficient data aggregation for distributed monitoring systems. *International Journal of Communication Systems, 31*(15), e3772, 2018.

[9] Benson, T., Anand, A., Akella, A., & Zhang, M. Microburst mitigation and traffic visibility in data centers. *IEEE Communications Magazine, 57*(5), 58 to 64, 2019.

[10] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. Scalable stream processing for continuous telemetry analytics. *ACM Transactions on Database Systems, 44*(1), 1 to 34, 2019.

[11] Ding, J., Chen, Q., & Xu, M. Memory efficient telemetry processing for distributed systems. *Journal of Systems Architecture, 98*, 121 to 132, 2019.

[12] Kumar, A., Singh, S., & Patel, D. Resource aware monitoring frameworks for cloud infrastructures. *IEEE Access, 7*, 156345 to 156357, 2019.

[13] Miao, R., Kim, M., & Rexford, J. Real time network telemetry using programmable switches. *IEEE Journal on Selected Areas in Communications, 37*(3), 499 to 513, 2019.

[14] Rao, P., Gupta, N., & Sharma, V. Efficient buffer management for large scale monitoring systems. *Computer Networks, 158*, 60 to 73, 2019.

[15] Zhao, L., Huang, T., & Liu, Y. Efficient congestion detection using distributed measurement aggregation. *Computer Networks, 162*, 106856, 2019.

[16] Zhang, S., Liu, X., & Wang, H. Runtime efficient monitoring in distributed cloud environments. *Future*

Generation Computer Systems, 103, 183 to 194, 2019.
[17] Chen, L., Liu, H., & Zhang, Y. Efficient telemetry collection for large scale cloud networks. *IEEE Transactions on Network and Service Management, 17*(4), 2398 to 2411, 2020.
[18] Gao, P., Narayan, A., & Stoica, I. Network telemetry for real time performance analysis in distributed systems. *ACM SIGCOMM Computer Communication Review, 50*(4), 29 to 41, 2020.
[19] Guo, C., Wu, H., Deng, Z., & Soni, A. High fidelity network monitoring with reduced overhead. *Proceedings of the IEEE International Conference on Network Protocols*, 233 to 244, 2020.
[20] Huang, Q., Birman, K., & Van Renesse, R. Scalable monitoring for cloud infrastructures using distributed aggregation. *Proceedings of the ACM Symposium on Cloud Computing*, 112 to 124, 2020.
[21] Jain, S., Kumar, A., & Mandal, S. Lightweight telemetry frameworks for resource efficient cloud monitoring. *Future Generation Computer Systems, 108*, 901 to 912, 2020.
[22] Lee, J., Park, H., & Kim, S. Reducing monitoring overhead in large scale distributed systems. *Computer Networks, 178*, 107347, 2020.
[23] Singh, R., Verma, P., & Joshi, A. Adaptive sampling for efficient runtime monitoring. *Journal of Systems and Software, 167*, 110596, 2020.
[24] Wang, L., Xu, Q., & Li, H. Processor and memory efficient telemetry analytics for distributed networks. *IEEE Access, 8*, 173845 to 173856, 2020.