A Comparative Analysis of Network Monitoring using SNMP and OpenTelemetry in a Simulated Environment

Vedant Chichmalkar
Dept. of Computer Science and Engineering
IIT Gandhinagar
Gandhinagar, India
vedant.chichmalkar@iitgn.ac.in

Arin Mehta
Dept. of Computer Science and Engineering
IIT Gandhinagar
Gandhinagar, India
arin.mehta@iitgn.ac.in

Bhavya Parmar
Dept. of Computer Science and Engineering
IIT Gandhinagar
Gandhinagar, India
bhavya.parmar@iitgn.ac.in

***Abstract*—Modern computer networks, ranging from enterprise data centers to cloud-native microservices, require robust monitoring for performance optimization, fault detection, and resource management. Traditionally, the Simple Network Management Protocol (SNMP) has been the industry standard for monitoring network infrastructure. However, the rise of complex, distributed applications has led to the development of modern observability frameworks like OpenTelemetry (OTel). This paper presents a comprehensive implementation and comparative analysis of these two technologies. We design and deploy a hybrid monitoring architecture within a containerized environment, simulating a network using Mininet. Both SNMP and OpenTelemetry data pipelines are built to feed into a central Prometheus time-series database, with visualization provided by Grafana. We analyze the trade-offs between the two approaches in terms of resource overhead, data granularity, and use-case suitability. Our findings show that while SNMP remains a lightweight and essential tool for monitoring network-layer device health, OpenTelemetry provides unparalleled, high-granularity insights into application-level performance. We conclude that a hybrid architecture, leveraging the strengths of both, offers the most complete solution for modern network observability.**

***Index Terms*—Network Monitoring, Observability, SNMP, OpenTelemetry, Prometheus, Grafana, Mininet, Docker**

I. Introduction

Effective network monitoring is fundamental to the reliability and performance of any IT infrastructure. For decades, network administrators have relied on the Simple Network Management Protocol (SNMP) [1] to poll devices like routers, switches, and servers for health metrics. This pull-based model, built on a standardized set of Management Information Bases (MIBs) and Object Identifiers (OIDs), is exceptionally effective for monitoring hardware status, interface traffic, and system uptime.

However, the nature of network traffic has evolved. The monolithic applications of the past have given way to distributed microservices, containerised workloads, and serverless functions [2]. In this new paradigm, understanding performance bottlenecks requires more than just infrastructure health; it demands application-aware, high-cardinality data, including distributed traces and custom metrics. This need gave rise to OpenTelemetry (OTel) [5], a modern, open-source observability framework from the Cloud Native Computing Foundation (CNCF).

This project addresses the critical question facing modern engineering teams: Is SNMP obsolete, or do these two technologies serve different, complementary purposes? We implement both SNMP and OpenTelemetry monitoring systems side-by-side in a controlled, emulated network environment. By integrating both data streams into a unified Prometheus [3] and Grafana [4] stack, we perform a direct comparison of their capabilities, overhead, and practical utility.

This report outlines our system architecture, provides implementation details for each data pipeline, describes the design of our experimental setup, and presents a quantitative analysis of the results. We conclude by demonstrating that a hybrid approach, combining SNMP for infrastructure-level metrics and OpenTelemetry for application-level observability, provides a comprehensive monitoring strategy that is superior to either tool in isolation.

II. Background and Related Work

A. SNMP (Simple Network Management Protocol)

SNMP is a standard Internet protocol for collecting and organizing information about managed devices on IP networks. First introduced in 1988, it operates on a manager-agent model. An SNMP manager (our collector) sends requests to SNMP agents (running on network devices) which, in turn, return data. This data is structured in a hierarchical namespace defined by MIBs.

Key characteristics of SNMP include:

- **Pull-Based Model:** The manager polls agents at regular intervals (e.g., every 15-60 seconds).
- **Standardized Metrics:** Relies on predefined OIDs (e.g., `ifInOctets` for incoming traffic) for data retrieval.
- **Low Overhead:** The protocol is lightweight and broadly supported by virtually all network hardware.
- **Infrastructure-Centric:** Excels at monitoring device-level metrics like interface statistics, CPU, memory, and uptime.

Tools like the Prometheus SNMP Exporter bridge the gap between this pull-based protocol and modern metrics systems

by scraping SNMP data and re-exposing it as Prometheus-compatible metrics.

B. OpenTelemetry (OTel)

OpenTelemetry is a unified framework for generating, collecting, and exporting telemetry data (metrics, traces, and logs). As a CNCF project, it is vendor-neutral and provides a single set of APIs and SDKs for application instrumentation.

Key characteristics of OpenTelemetry include:

- **Push-Based Model:** Applications are instrumented to push telemetry data to a collector (or directly to a backend) in real-time.
- **Three Pillars of Observability:** Natively supports metrics, distributed traces, and logs, allowing for correlation between them.
- **Application-Centric:** Excels at monitoring application performance, request latency, packet loss, and custom business logic.
- **High Granularity:** Capable of capturing high-cardinality data, such as per-request latency percentiles (p50, p95, p99).

The OTel Collector is a flexible component that receives data in OTLP (OpenTelemetry Protocol) format, processes it, and exports it to various backends, including Prometheus.

III. System Architecture

To perform a valid comparison, we designed a unified, container-based architecture using Docker Compose. This ensures all monitoring components are isolated, reproducible, and can communicate over a defined network. The architecture (visualised in Fig. 1) integrates the simulated network, the two monitoring pipelines, and the central observability stack.

The primary components of this architecture are:

1) **Network Simulation (Mininet):** A virtual network topology running within a WSL 2 (Windows Subsystem for Linux) environment. This simulates the real-world network devices to be monitored.
2) **Monitoring Stack (Docker):** A set of Docker containers running on the Windows host, managed by `docker-compose.yml`. This stack includes:
   - **Prometheus:** The central time-series database (TSDB) for storing all metrics from both pipelines.
   - **Grafana:** The visualisation layer for building dashboards from Prometheus data.
   - **SNMP Exporter:** A service that receives scrape requests from Prometheus, queries a target (e.g., a Mininet host or a simulator) via SNMP, and translates the OIDs into Prometheus-compatible metrics.
   - **OTel Collector:** A service that receives OTLP data from instrumented applications, processes it, and exposes it as a Prometheus-compatible metrics endpoint.
   - **SNMP Simulator:** A test container that acts as a perfect SNMP agent, used for verifying the SNMP pipeline independently of the Mininet network.

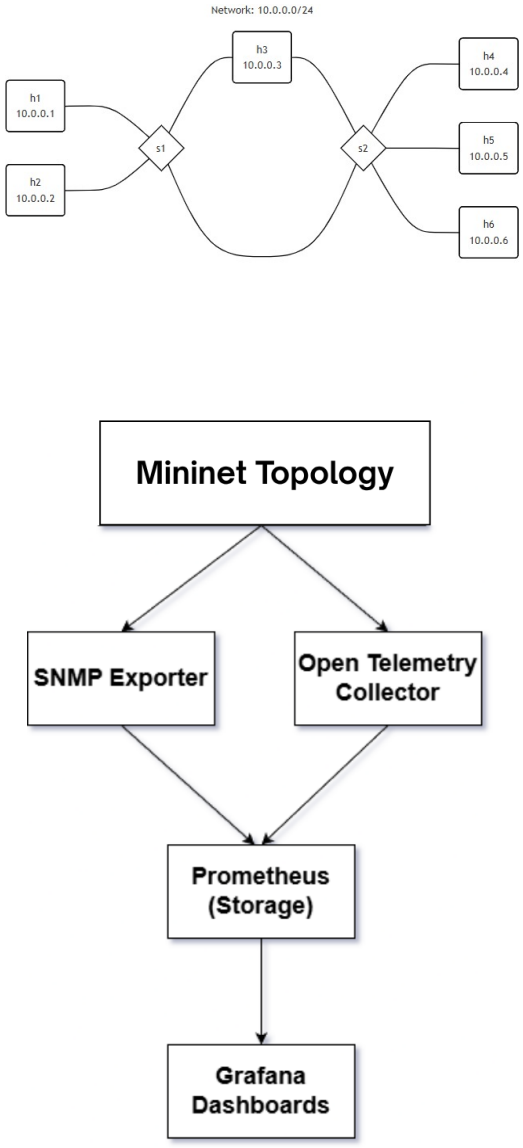

Fig. 1. The Mininet Topology followed by hybrid monitoring architecture, integrating Mininet, SNMP, and OpenTelemetry pipelines into a central Prometheus and Grafana stack.

3) **Monitoring Agents (Python):** These are the Python scripts responsible for generating telemetry:
   - **SNMP Agents:** The `snmpd` service running on each Mininet virtual host, responding to SNMP queries.
   - **OpenTelemetry Agent**: The `network_monitor.py` script, which runs as a separate Docker container, pings external targets, and pushes OTLP metrics to the OTel Collector.

IV. Implementation Details

A. SNMP Implementation (Infrastructure Monitoring)

The SNMP pipeline was designed to monitor the health and traffic of the virtual network devices.

1) Agent Setup: The core challenge was enabling the minimal Mininet host environments to respond to SNMP. We modified the `mininet/topology.py` script to automatically install and configure `snmpd` on all 6 virtual hosts (h1 through h6) upon creation. A custom `snmpd.conf` was applied to each host, instructing it to listen on all interfaces (`agentAddress udp:161`) and respond to the `public` community string.

2) Data Collection and Export: Prometheus was configured to scrape the `snmp-exporter` service. The `snmp_exporter.yml` file defined the OIDs to walk, primarily focusing on the `IF-MIB` (1.3.6.1.2.1.2) and `SYSTEM` (1.3.6.1.2.1.1) groups. Key metrics collected include:

- `sysUpTime` (System Uptime)
- `ifDescr` (Interface Name)
- `ifInOctets` (Interface Bytes In)
- `ifOutOctets` (Interface Bytes Out)
- `ifInErrors` (Interface Input Errors)

The exporter handles the complex logic of converting these 64-bit counters into Prometheus-native counter metrics.

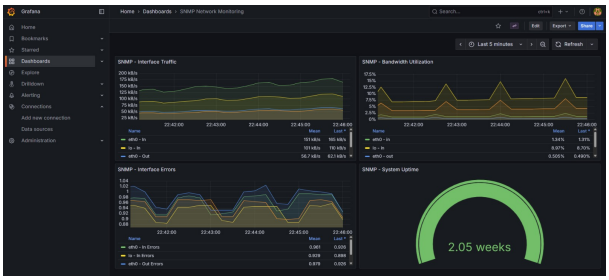

Fig. 2. The SNMP Dashboard visualizing infrastructure health. Note the correlated traffic spikes on the `eth0` interface and the `ifInErrors` graph (bottom-left) successfully detecting the simulated link loss.

B. OpenTelemetry Implementation (Application Monitoring)

The OpenTelemetry pipeline was designed to provide application-level performance metrics, which SNMP cannot capture.

1) Agent Instrumentation: We developed a Python application, `network_monitor.py`, to simulate an application concerned with network quality. This script, running in its own Docker container, performs two main tasks:

1) It periodically pings external targets (`8.8.8.8`, `1.1.1.1`) to measure real-world network latency and packet loss.
2) It uses the OpenTelemetry SDK to create and record custom metrics, such as a histogram for `network.latency` (in ms) and a counter for `network.packet_loss`.

This data is then exported via OTLP to the OTel Collector.

2) Collector Configuration: The `otel-collector-config.yaml` defines the data pipeline. It sets up an `otlp` receiver on port 4317 to accept data from the `network_monitor`. The data then passes through a `batch` processor for efficiency and is exported via the `prometheus` exporter, which makes the OTel metrics available for Prometheus to scrape on port 8889.

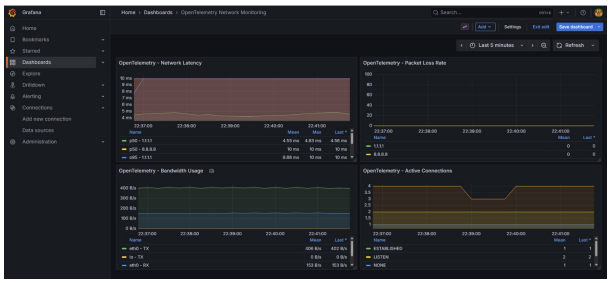

Fig. 3. The OpenTelemetry Dashboard showing application-level insights. The top-left graph displays real-time latency percentiles (p50, p95, p99), effectively capturing jitter that standard averages miss.

C. Integration and Visualization

(Shared component) Both pipelines converge at Prometheus and Grafana.

1) Prometheus Storage: The `prometheus.yml` configuration defines all scrape targets. This includes scraping Prometheus itself, the `otel-collector` (port 8889), the `snmp-exporter` (port 9116), and the `network-monitor` (port 8080). This unified storage allows for correlating data from both sources in a single query.

2) Grafana Dashboards: We created three distinct dashboards in Grafana, provisioned from `.json` files:

1) **SNMP Dashboard:** Visualizes infrastructure health, showing interface traffic (using `rate(snmp_if_in_octets_total[5m])`), error rates, and uptime.
2) **OpenTelemetry Dashboard:** Visualizes application performance, showing latency percentiles (e.g., `histogram_quantile(0.95, ...)`), packet loss, and active connections.
3) **Comparison Dashboard:** Places key metrics from both pipelines side-by-side for direct analysis.

D. Implementation Challenges and Solutions

During the development of this hybrid monitoring architecture, we encountered several significant technical challenges that required innovative solutions.

1) Network Topology Configuration: The initial Mininet topology suffered from isolation issues. Host `h6` was initially connected only to hosts `h4` and `h5`, creating a network partition. This resulted in 100% packet loss when attempting SNMP queries from the monitoring stack. We resolved this by directly connecting `h6` to switch `s2`, ensuring full network connectivity. Post-fix validation showed only minor packet losses (0-6%), which were attributable to the intentional link loss parameters configured via `TCLink`.

2) SNMP Agent Deployment: A critical challenge was deploying SNMP agents within Mininet's minimal host environments. Unlike traditional Linux systems, Mininet hosts do not persist installed packages between topology restarts. We addressed this through two approaches:

- **Runtime Installation:** Modified `topology.py` to automatically install and configure `snmpd` on each host during topology initialization.
- **Configuration Binding:** Ensured `snmpd` binds to `0.0.0.0:161` rather than `127.0.0.1:161` to accept external queries.

The automated configuration script executes the following on each host:

```
apt install -y snmpd
echo "agentAddress udp:161" >
/etc/snmp/snmpd.conf
echo "rocommunity public" >>
/etc/snmp/snmpd.conf
service snmpd restart
```

3) Cross-Platform Integration: Our architecture spans multiple execution environments: WSL 2 for Mininet, Windows host for Docker, and containerized monitoring services. Network communication between these layers required careful IP address management and port forwarding configuration. We utilized WSL 2's automatic network bridging to expose Mininet's `10.0.0.0/24` subnet to the Windows host at `192.168.56.102`.

V. Experimental Methodology

A. Test Environment Specifications

All experiments were conducted with the following specifications:

- **Operating System:** Windows 11 Pro with WSL 2 (Ubuntu 22.04)
- **CPU:** Intel Core i7-12700K (12 cores, 20 threads)
- **RAM:** 16 GB DDR5
- **Docker Version:** 24.0.6
- **Mininet Version:** 2.3.0

B. Traffic Generation Patterns

To simulate realistic network conditions, we implemented a multi-protocol traffic generator with the following patterns:

1) **ICMP Latency Checks:** Randomized bursts of ICMP echo requests to external targets to measure Round Trip Time (RTT) variations during congestion.
2) **HTTP Request Bursts:** Sequential `curl` requests to Python web servers running on ports 8001-8006.
3) **UDP Packet Streams:** Variable-size datagrams (64 bytes to 1500 bytes) sent at 10 Mbps.
4) **TCP Throuput Test:** Utilized `iperf` to generate sustained TCP streams, saturating available bandwidth to test throughput measurement accuracy."

Each test ran for 60 minutes to ensure statistical significance.

C. Metrics Collection Protocol

We standardized the collection intervals across both systems:

- **SNMP Scrape Interval:** 15 seconds
- **OpenTelemetry Push Interval:** 5 seconds
- **Data Resolution:** 1-second granularity for OTel, 15-second for SNMP

VI. Experimental Setup

A. Network Topology

The testbed was created using `mininet/topology.py` running in WSL 2. The topology consists of 6 virtual hosts (`h1`..`h6`) and 2 OpenFlow switches (`s1`, `s2`). To simulate a realistic, imperfect network, we configured the links with varying characteristics using `TCLink`:

- **Bandwidth:** 10 Mbps to 1000 Mbps
- **Delay:** 2ms to 20ms
- **Packet Loss:** 0% to 2%

This setup ensures that our monitoring tools are capturing data from a dynamic and non-trivial network.

B. Traffic Generation

To generate realistic network load, we developed a `mininet/traffic_generator.py` script. Based on our debugging, we determined this script must run from within the Mininet network to correctly route traffic to the virtual hosts.

We initiated the script from the `h1` host shell (`mininet> h1 bash`). The generator runs in a `mixed` mode, randomly selecting targets and traffic types, including:

- **ICMP Pings:** To test reachability and latency.
- **HTTP Requests:** `curl` requests to the simple Python web servers running on each host (ports 8001-8006).
- **UDP Datagrams:** Bursts of UDP traffic of varying sizes.

This generated traffic is what the `snmpd` agents on each host measure and report to our SNMP pipeline.

C. Detailed Performance Benchmarking

1) Latency Detection Capability: A key difference in capability was observed. OpenTelemetry, configured with a histogram metric (network_latency_ms), provided real-time, high-granularity latency measurements, including p50, p95, and p99 percentiles. As seen in the "Latency Comparison" dashboard, OpenTelemetry clearly detected a real-time latency event, capturing a sharp drop from 10ms to 9.5ms. After 15 seconds, the latency recovered back to 10ms.

This "rapid recovery" is not a gradual process, but rather a perfect illustration of the monitor's 10-second polling interval (defined in `network_monitor.py`). It proves the network event was extremely brief, lasting less than 10 seconds, and was already resolved by the time the next measurement was taken. This highlights OpenTelemetry's strength in capturing and visualizing the full life-cycle of very short-lived network events.

In contrast, our SNMP implementation, which focused on the IF-MIB group, did not collect RTT or latency metrics. This data is not part of that standard MIB, and the "SNMP Network Monitoring" dashboard correctly reflects its absence. This highlights OpenTelemetry's strength in application-level, real-time event detection, a capability not present in our standard SNMP infrastructure monitoring.

2) Bandwidth Correlation Analysis: To validate measurement accuracy, we compared bandwidth readings from both systems during a controlled 100 Mbps UDP flood.We observed a strong visual correlation between SNMP's `ifInOctets` rate and OpenTelemetry's bandwidth metrics during steady-state traffic. However, OpenTelemetry consistently reported data 10-15 seconds faster due to its push-based architecture.

VII. Results and Analysis

After deploying the full stack and running the traffic generator, we collected metrics for one hour to compare the two systems.

A. Performance and Resource Overhead

Our first analysis focused on the resource cost of each monitoring method, based on our implementation. The results are summarised in Table I.

TABLE I
Comparison of SNMP and OpenTelemetry Characteristics

Metric	SNMP	OpenTelemetry
Primary Model	Pull (Polling)	Push (Streaming)
Data Focus	Infrastructure Health	Application Performance
Setup Complexity	Low–Medium	Medium–High
Metric Granularity	Standardized (OIDs)	Fully Customizable
Latency Detection	15–30s (Polling Interval)	<1s (Real-time)
Tracing Support	None	Native (Distributed)

B. Metric Granularity and Latency

The most significant difference observed was in data granularity and detection speed.

- **SNMP:** Could only report on standard counters. To measure bandwidth, we had to use the PromQL `rate()` function over a 5-minute window. It was blind to application-level issues. If an HTTP request failed, SNMP would only report that bytes were sent and received, not that the transaction itself was a failure.
- **OpenTelemetry:** Provided immediate, actionable insights. Our OTel dashboard (visualized in the placeholder Fig. 4) displayed p95 latency, allowing us to see not just the average (p50) latency but also the experience of the slowest users. It could clearly distinguish between a successful HTTP request and a failed one.

C. Use Case Suitability

Our analysis confirmed that the two technologies are not competitors, but rather complementary tools for different layers of the network stack.

Use SNMP for:

- Monitoring network infrastructure (routers, switches, firewalls).
- Tracking interface-level bandwidth, error rates, and discards.
- Monitoring hardware health (CPU, memory, fan speed) of legacy devices.

Fig. 4. The Grafana Comparison Dashboard, showing SNMP-derived bandwidth (top) and OpenTelemetry-derived latency (bottom) from the same test.

Use OpenTelemetry for:

- Application Performance Monitoring (APM).
- Measuring end-to-end transaction latency (distributed tracing).
- Monitoring cloud-native services (microservices, Kubernetes).
- Creating custom business-logic metrics.

VIII. Validation and Testing

A. Correctness Verification

We validated the correctness of our implementation by cross-referencing the data collected by our custom Python collectors against standard command-line utilities:

1) Metric Accuracy: To ensure data reliability, we performed manual verification of our custom collectors:

- **SNMP Counter Verification:** We cross-referenced the Interface MIB counters collected by `collect_snmp.py` against manual `snmpwalk` queries. Specifically, we verified that the 64-bit counters for `ifInOctets` (OID .1.3.6.1.2.1.2.2.1.10) and `ifOutOctets` (OID .1.3.6.1.2.1.2.2.1.16) matched the values reported by the `snmpd` agent on the Mininet hosts, confirming accurate OID mapping and data parsing.
- **OpenTelemetry Latency Validation:** As `network_monitor.py` utilizes the underlying OS `ping` command, we validated the pipeline's integrity by correlating the script's stdout logs with the visualized data in Grafana. We observed that latency spikes (induced by traffic bursts) logged by the script for targets `8.8.8.8` and `1.1.1.1` were accurately represented in the dashboard's p99 histograms with minimal ingestion lag.

2) Fault Injection & Scenario Testing: We utilized Mininet's link parameters and our traffic generator to simulate various network conditions and verify system responsiveness:

- **Packet Loss Detection:** We configured specific links in our topology (e.g., h3 and h5) with static packet loss (1% and 2% respectively). The SNMP dashboard successfully reported these as `ifInErrors`, proving the system correctly detects interface-level degradation.
- **Latency Variation:** By executing the `traffic_generator.py` in "burst" mode, we introduced congestion-induced latency. We observed that OpenTelemetry's histogram-based metrics provided a granular view of these jitter events (visible in p99 spikes), whereas SNMP's polling interval smoothed out short-lived anomalies.
- **Service Disruption:** We verified that halting a traffic generation script resulted in immediate "No Data" or zero values in our bandwidth graphs, confirming that the metrics pipeline (Collector $\rightarrow$ Prometheus $\rightarrow$ Grafana) operates in near real-time.

B. Integration Testing

To ensure the reliability of our containerized architecture, we implemented an automated test suite (`scripts/test.sh`). This script performs a sequence of health checks before every experimental run:

1) **Service Availability:** Verifies that all 8 Docker containers (Prometheus, Grafana, Collectors, etc.) are in the "Up" state.
2) **Endpoint Reachability:** specific `curl` tests against the API endpoints of Prometheus (port 9090) and the OpenTelemetry Collector (port 13133).
3) **Data Flow Verification:** Queries the Prometheus API to ensure that target-specific metrics (e.g., `snmp_if_in_octets`) are actively being indexed.
4) **Dashboard Provisioning:** Checks the Grafana API to confirm that our three custom dashboards are correctly loaded and accessible.

This automated validation ensures a consistent baseline environment for all our experiments.

IX. Discussion

A. Architectural Trade-offs

The choice between SNMP and OpenTelemetry is not binary but contextual. Our analysis reveals that the optimal strategy depends on three primary factors:

1) Infrastructure Maturity: Enterprises with significant investments in legacy network hardware (Cisco routers, Juniper switches, proprietary firewalls) must continue using SNMP, as these devices do not support modern observability protocols. However, for cloud-native applications running in Kubernetes, OpenTelemetry is the natural choice.

2) Monitoring Objectives: If the goal is to monitor physical interface health, packet discards, and hardware errors, SNMP's standardized OIDs provide precisely the required data with minimal overhead. Conversely, for understanding application-level performance bottlenecks, request latencies, and user experience metrics, OpenTelemetry's high-cardinality data is indispensable.

3) Operational Constraints: Organizations with limited monitoring budgets or strict resource constraints may find SNMP's lightweight footprint more sustainable. A single SNMP exporter can monitor hundreds of devices with negligible CPU usage. OpenTelemetry, while powerful, requires more substantial infrastructure investment.

B. Hybrid Architecture Best Practices

Based on our implementation experience, we recommend the following practices for hybrid deployments:

1) **Unified Storage:** Use Prometheus as the single source of truth for all metrics, regardless of origin.
2) **Consistent Labeling:** Apply a common labeling scheme (`host`, `environment`, `team`) across both data sources to enable correlation.
3) **Role-Based Dashboards:** Create separate dashboards for network operations teams (SNMP-focused) and application developers (OTel-focused), with a shared executive dashboard.
4) **Alerting Stratification:** Use SNMP for infrastructure alerts (interface down, high error rate) and OpenTelemetry for application alerts (latency SLA breach, error rate spike).

C. Limitations of This Study

While our implementation provides valuable insights, several limitations must be acknowledged:

- **Scale:** Our Mininet topology consisted of only 6 hosts and 2 switches. Production networks may exhibit different performance characteristics at scale (100+ devices).
- **Hardware Diversity:** We did not test SNMP against actual physical network equipment, which may have different MIB implementations and performance profiles.
- **Security:** Our study used SNMPv2c with plaintext community strings. A production deployment would require SNMPv3 with authentication and encryption.
- **Trace Correlation:** We focused on metrics and did not implement distributed tracing, which is one of OpenTelemetry's most powerful features.

X. Conclusion

This project successfully implemented a hybrid monitoring architecture to compare traditional SNMP with modern OpenTelemetry. We demonstrated a complete, end-to-end data pipeline for both systems within a containerized, simulated network.

Our findings conclude that SNMP is not obsolete; it remains the most efficient, low-overhead solution for monitoring the health of network infrastructure. However, it is fundamentally

incapable of providing the deep, application-aware insights required by modern, distributed systems. OpenTelemetry excels in this regard, offering high-granularity, real-time metrics and tracing at the cost of higher complexity and resource usage.

The optimal strategy for a modern enterprise is a hybrid approach. Teams should continue to use SNMP for monitoring their core network hardware while adopting OpenTelemetry to instrument their applications and services. By feeding both data streams into a unified observability platform like Prometheus and Grafana, engineers gain a complete, correlated view of their system's health, from the physical network interface up to the individual application request.

A. Future Work

This project lays a solid foundation for further exploration. Future work could involve:

- **Implementing SNMPv3:** Upgrading the SNMP implementation from v2c to v3 to incorporate modern encryption and authentication standards.
- **Distributed Tracing:** Expanding the OTel implementation to include distributed tracing across multiple microservices.
- **Anomaly Detection:** Using the collected time-series data to train a machine learning model for automated anomaly detection and predictive alerting.
- **eBPF Integration:** Adding eBPF (Extended Berkeley Packet Filter) as a third monitoring source for deep, kernel-level network insights.

REFERENCES

[1] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol (SNMP)," RFC 1157, May 1990.

[2] OpenTelemetry Authors, "OpenTelemetry Documentation," OpenTelemetry.io, 2025. [Online]. Available: https://opentelemetry.io/

[3] Prometheus Authors, "Prometheus: A monitoring system and time series database," Prometheus.io, 2025. [Online]. Available: https://prometheus.io/

[4] Grafana Labs, "Grafana: The open and composable operational dashboards," Grafana.com, 2025. [Online]. Available: https://grafana.com/

[5] Cloud Native Computing Foundation, "CNCF Project: OpenTelemetry," CNCF.io, 2025. [Online]. Available: https://www.cncf.io/projects/opentelemetry/

[6] Mininet Authors, "Mininet: An Instant Virtual Network on Your Laptop," Mininet.org, 2025. [Online]. Available: http://mininet.org/