School of Computer Science and Information Technology

Department of Computer Science and Information Technology

Semester: IV
Specialisation: Internet of Things (IoT)

23BCA4VC02: Network Administration

Activity 2

NETWORK MONITORING TOOL

Date of Submission: 03-05-2025

Submitted by:	**Faculty In-Charge:**
Name: Arpit Sarkar	**Mr. Sahabzada Betab Badar**
Reg No./USN No: 23BCAR0313	
Signature:	

INDEX

GitHub - https://github.com/exoduusss/Network-Monitor

LinkedIn-https://www.linkedin.com/posts/arpit-sarkar-a1b1ba299_networkmonitoring-iot-machinelearning-activity-7324375357795487744-zWlP?utm_source=share&utm_medium=member_desktop&rcm=ACoAAEgcYDcB4eYmIIcwI6SW2SCo3xhs76h5Vtw

INTRODUCTION

This **Network Monitor** is a versatile tool designed to help you keep track of the status and performance of multiple servers or websites in real time. It combines various features to ensure reliable and insightful monitoring, including:

- **Multi-threaded Monitoring**: Simultaneously check the status of multiple targets for efficiency.
- **Data Visualization**: Real-time graphs and map visualizations for response times and server locations.
- **Anomaly Detection**: Identify unusual patterns in response times using machine learning.
- **Geolocation Tracking**: Retrieve and display server location data.
- **REST API**: Expose monitoring data through a web API for easy integration with other tools.
- **Database Logging**: Maintain a historical log of checks with detailed data using SQLite.
- **User Alerts**: Notify users of consecutive failures for proactive issue management.

Background

In today's digitally connected world, businesses and individuals rely heavily on the continuous availability and performance of servers and online services. A single failure in a critical service can lead to financial losses, damaged reputations, and dissatisfied users.

Monitoring the health of websites, servers, and network endpoints is essential for:

1. **Ensuring Uptime**: Detecting and addressing service outages promptly minimizes downtime and its associated costs.
2. **Performance Optimization**: Tracking response times helps identify bottlenecks and maintain optimal performance.
3. **Security Awareness**: Monitoring patterns can provide early warnings of potential security threats, such as distributed denial-of-service (DDoS) attacks.
4. **Resource Allocation**: Data from monitoring enables better planning for scaling and infrastructure improvements.

Importance

This project bridges the gap by offering a comprehensive **Network Monitor** tool that incorporates state-of-the-art features and customization capabilities. Its importance can be summarized as follows:

1. **Proactive Problem Detection**: Alerts and anomaly detection ensure issues are addressed before they escalate.
2. **Enhanced Decision-Making**: Historical data and visual insights empower users to make informed decisions about network infrastructure and optimizations.
3. **Improved User Experience**: By maintaining high uptime and fast response times, businesses can offer better experiences to their customers.
4. **Scalable Design**: Whether for small-scale personal use or enterprise-level monitoring, the tool adapts to various requirements.
5. **Convenient Integration**: The REST API allows easy integration with existing tools and workflows, making it versatile for developers and IT teams.

Challenges:

Real-Time Data Handling: Ensuring efficient processing of real-time data without delays or bottlenecks.

1. **Scalability**: Designing a system that can handle both small and large-scale networks seamlessly.
2. **Accurate Anomaly Detection**: Implementing reliable algorithms to minimize false positives and negatives.
3. **Security**: Protecting sensitive data and ensuring secure system access.
4. **Cross-Platform Compatibility**: Making the tool functional across various operating systems.
5. **User-Friendly Design**: Balancing powerful features with an intuitive interface for ease of use.

IMPLEMENTATION

How It Works:

1. **Setup and Initialization**: Users specify targets, monitoring intervals, and alert thresholds. The tool initializes databases, anomaly detection models, and visualizations.
2. **Monitoring Loop**: For each target, the tool performs:
   - **Status Checks**: Monitors uptime, response time, and HTTP status codes.
   - **DNS Caching and Geolocation**: Retrieves server IP addresses and location data to enhance monitoring context.
   - **Data Logging**: Logs results to an SQLite database and updates historical records.
3. **Real-Time Updates**: Visualizes monitoring results with plots of response times and geographic mapping.
4. **Anomaly Detection**: Identifies performance anomalies based on response time trends.
5. **REST API Access**: Exposes monitoring data via a Flask-based API for external access.
6. **Reporting**: Provides periodic summaries and access to detailed data through the API.

1. Class Initialization and Setup

```
class AdvancedNetworkMonitor:
    def __init__(self, targets, interval=60, alert_threshold=3, api_port=5000):
        self.targets = targets
        self.interval = interval
        self.alert_threshold = alert_threshold
        self.api_port = api_port
        self.failure_counts = {target: 0 for target in targets}
        self.history = {target: {'timestamps': [], 'response_times': [], 'status': [],
                        'status_codes': [], 'locations': []} for target in targets}
        self._dns_cache = {}
        self.anomaly_detectors = {target: IsolationForest(contamination=0.05) for
target in targets}

        self.setup_database()
        self.setup_plots()
        self.start_api()
```

2. SQLite Database Setup

```
def setup_database(self):
    """Initialize SQLite database with proper datetime handling"""
    def adapt_datetime(dt):
        return dt.isoformat()

    def convert_datetime(text):
        return datetime.fromisoformat(text.decode())

    sqlite3.register_adapter(datetime, adapt_datetime)
    sqlite3.register_converter("datetime", convert_datetime)

    self.conn = sqlite3.connect('network_monitor.db',
detect_types=sqlite3.PARSE_DECLTYPES)
    self.cursor = self.conn.cursor()
    self.cursor.execute('''
        CREATE TABLE IF NOT EXISTS status_checks (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            timestamp DATETIME,
            target TEXT,
            status TEXT,
            response_time REAL,
            status_code INTEGER,
            country TEXT,
            city TEXT
        )
    ''')
    self.conn.commit()
```

3. Status Check

```
def check_status(self, url):
    """Enhanced status check with DNS caching and geolocation"""
    try:
        parsed = urlparse(url)
        if not parsed.scheme:
            url = 'http://' + url
            parsed = urlparse(url)

        if parsed.netloc not in self._dns_cache:
            self._dns_cache[parsed.netloc] = socket.gethostbyname(parsed.netloc)
        location = self.get_geolocation(url)

        start_time = time.time()
        response = requests.get(url, timeout=5, allow_redirects=True,
headers={'User-Agent': 'NetworkMonitor/3.0'})
        response_time = (time.time() - start_time) * 1000

        status = 'UP' if response.status_code < 400 else 'DEGRADED'
        return status, response_time, response.status_code, location
    except requests.exceptions.Timeout:
        return 'TIMEOUT', 0, 408, None
    except requests.exceptions.SSLError:
        return 'SSL_ERROR', 0, 495, None
    except requests.exceptions.ConnectionError:
        return 'CONN_ERROR', 0, 503, None
    except Exception as e:
        return 'ERROR', 0, str(e), None
```

4. Geolocation

```
def get_geolocation(self, url):
    """Get server location information"""
    try:
        domain = urlparse(url).netloc
        ip = self._dns_cache.get(domain, socket.gethostbyname(domain))
        response = requests.get(f'http://ip-
api.com/json/{ip}?fields=country,city,lat,lon,isp').json()
        return {
            'country': response.get('country'),
            'city': response.get('city'),
            'lat': response.get('lat'),
            'lon': response.get('lon'),
            'isp': response.get('isp')
        }
    except:
        return None
```

5. Anomaly Detection

```
def detect_anomalies(self, target):
    """Machine learning anomaly detection on response times"""
    if len(self.history[target]['response_times']) < 10:
        return []

    X = np.array(self.history[target]['response_times']).reshape(-1, 1)
    self.anomaly_detectors[target].fit(X)
    anomalies = self.anomaly_detectors[target].predict(X)
    return [i for i, x in enumerate(anomalies) if x == -1]
```

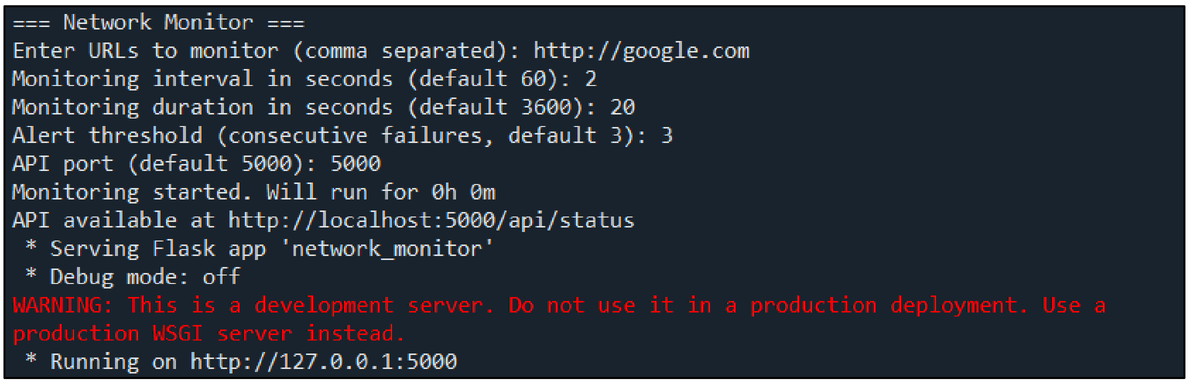
6. Flask API

```
def start_api(self):
    """Start Flask API in a separate thread"""
    app = Flask(__name__)

    @app.route('/api/status')
    def get_status():
        return jsonify({
            'targets': self.targets,
            'current_status': {t: self.history[t]['status'][-1] if self.history[t]['status']
            else None
                        for t in self.targets},
            'stats': self.get_stats()
        })

    @app.route('/api/history/<target>')
    def get_history(target):
        if target not in self.history:
            return jsonify({'error': 'Target not found'}), 404
        return jsonify(self.history[target])

    @app.route('/api/anomalies')
    def get_anomalies():
        return jsonify({t: self.detect_anomalies(t) for t in self.targets})

    Thread(target=lambda: app.run(port=self.api_port), daemon=True).start()
```

7. Running the Monitor

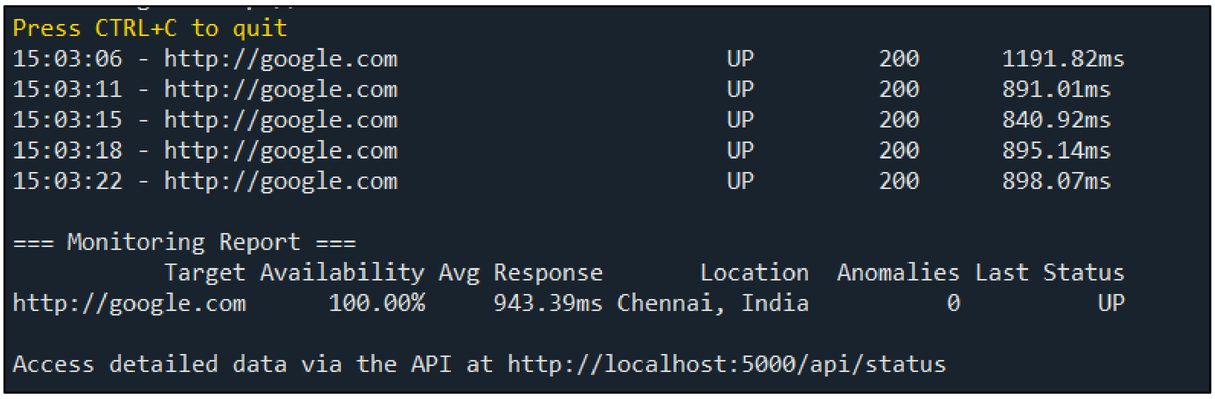
```
def run_monitor(self, duration=3600):
    """Enhanced monitoring loop with all features"""
    end_time = time.time() + duration
    print(f"Monitoring started. Will run for {duration//3600}h
{duration%3600//60}m")
    print(f"API available at http://localhost:{self.api_port}/api/status")

    try:
        with concurrent.futures.ThreadPoolExecutor() as executor:
            while time.time() < end_time:
                futures = {executor.submit(self.check_status, target): target for target
in self.targets}

                for future in concurrent.futures.as_completed(futures):
                    target = futures[future]
                    status, response_time, code, location = future.result()
                    timestamp = datetime.now().strftime("%H:%M:%S")
                    self.history[target]['timestamps'].append(timestamp)
                    self.history[target]['response_times'].append(response_time)
                    self.history[target]['status'].append(status)
                    self.history[target]['status_codes'].append(code)
                    self.history[target]['locations'].append(location)
                    self.log_to_database(target, status, response_time, code, location)

                    print(f"{timestamp} - {target:40} {status:10} {str(code):8}
{response_time:6.2f}ms")

                self.update_plots()
                time.sleep(self.interval)

    except KeyboardInterrupt:
        print("\nMonitoring stopped by user")
    finally:
        self.generate_report()
        plt.ioff()
        plt.show()
        self.conn.close()
```

SCREENSHOTS



```
=== Network Monitor ===
Enter URLs to monitor (comma separated): http://google.com
Monitoring interval in seconds (default 60): 2
Monitoring duration in seconds (default 3600): 20
Alert threshold (consecutive failures, default 3): 3
API port (default 5000): 5000
Monitoring started. Will run for 0h 0m
API available at http://localhost:5000/api/status
 * Serving Flask app 'network_monitor'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
 * Running on http://127.0.0.1:5000
```

Fig 1: Console Output Overview



```
Press CTRL+C to quit
15:03:06 - http://google.com                          UP          200       1191.82ms
15:03:11 - http://google.com                          UP          200       891.01ms
15:03:15 - http://google.com                          UP          200       840.92ms
15:03:18 - http://google.com                          UP          200       895.14ms
15:03:22 - http://google.com                          UP          200       898.07ms

=== Monitoring Report ===
           Target Availability Avg Response        Location  Anomalies Last Status
http://google.com      100.00%     943.39ms Chennai, India          0          UP

Access detailed data via the API at http://localhost:5000/api/status
```

Fig 2: Live Monitoring and Report Summary

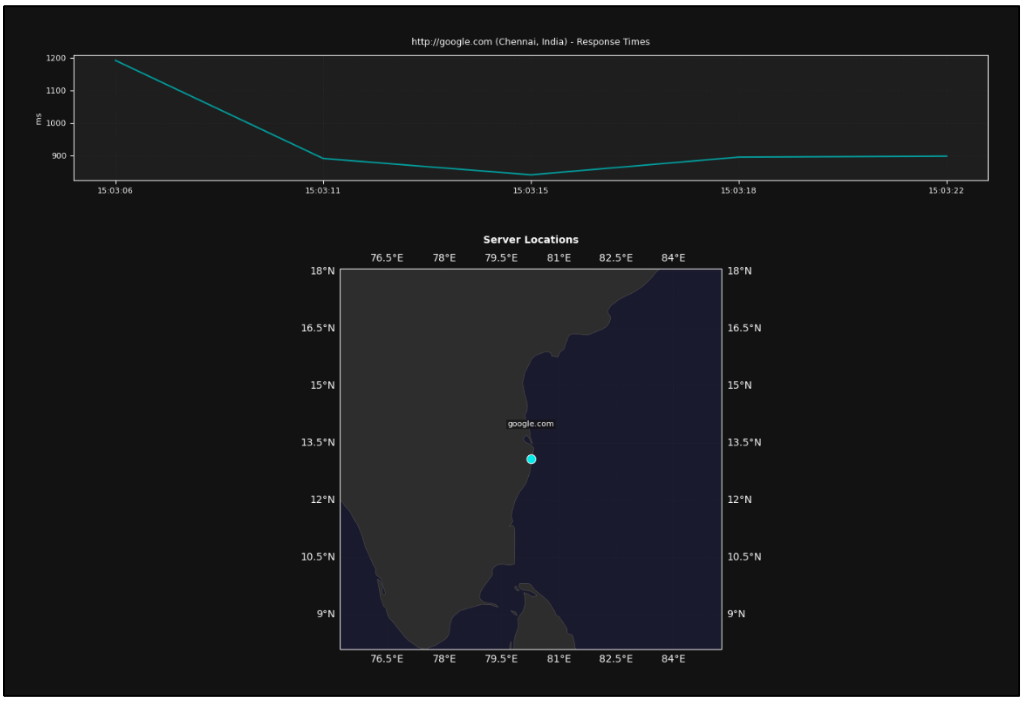
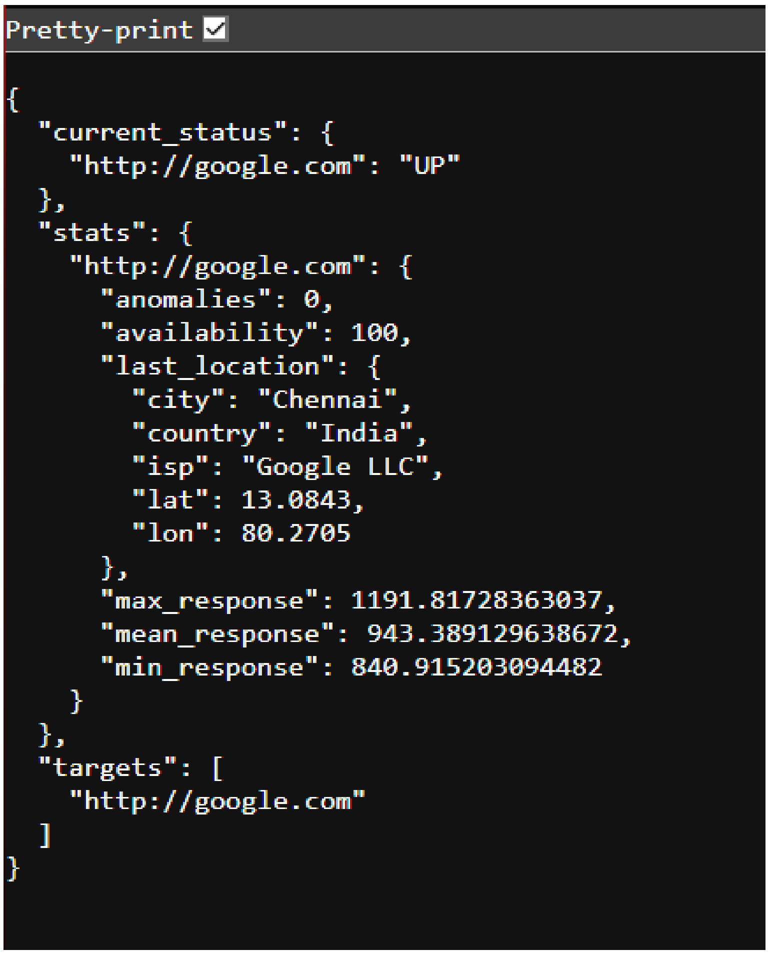


Fig: Visualization of Response Times and Server Geolocation



Pretty-print ✓

```
{
  "current_status": {
    "http://google.com": "UP"
  },
  "stats": {
    "http://google.com": {
      "anomalies": 0,
      "availability": 100,
      "last_location": {
        "city": "Chennai",
        "country": "India",
        "isp": "Google LLC",
        "lat": 13.0843,
        "lon": 80.2705
      },
      "max_response": 1191.81728363037,
      "mean_response": 943.389129638672,
      "min_response": 840.915203094482
    }
  },
  "targets": [
    "http://google.com"
  ]
}
```

Fig: REST API Overview

CONCLUSION

The **Network Monitor** provides a powerful tool for monitoring the status and performance of multiple websites or services. With features like real-time status tracking, response time analysis, anomaly detection using machine learning, and geolocation mapping, it offers a comprehensive solution for ensuring the health of your network. The integration of a database for logging, an API for remote access, and dynamic visualizations makes it easy to track and analyze network performance over time. This tool helps in identifying issues early, improving uptime, and providing detailed reports for better decision-making in network management.