Juniper Topology

Mist REST API Network Topology Retrieval Guide

Juniper Mist provides comprehensive REST API support for network topology discovery outside EVPN environments, with dedicated endpoints for traditional switching architectures, device discovery, and connectivity mapping. This guide covers the complete implementation approach for retrieving non-EVPN topology information through the Mist Cloud API using efficient bulk retrieval strategies.

Available Mist REST API endpoints for topology retrieval

The Mist REST API offers multiple endpoint categories for comprehensive topology discovery in non-EVPN environments. **Regional API endpoints vary by organization location** - replace `manage` with `api` in your portal URL to determine the correct endpoint (e.g., `api.mist.com`, `api.eu.mist.com`, `api.ac2.mist.com`).

Organization-level bulk endpoints (Most Efficient)

These endpoints retrieve all data across your entire organization in single API calls, dramatically reducing rate limit consumption:

- `GET /api/v1/orgs/{org_id}/inventory` - **Complete device inventory across all sites** (single call for everything)
- `GET /api/v1/orgs/{org_id}/stats/devices` - Organization-wide device statistics with connectivity details
- `GET /api/v1/orgs/{org_id}/devices/search` - Search and filter devices across the organization
- `GET /api/v1/orgs/{org_id}/insights/switch-metrics` - Bulk switch metrics and topology insights
- `GET /api/v1/orgs/{org_id}/devices/export` - Export complete device configurations

Site-level endpoints (Use only when necessary)

Use these endpoints only when you need specific site-level detail that isn't available in bulk:

- `GET /api/v1/sites/{site_id}/devices` - Lists devices at a specific site
- `GET /api/v1/sites/{site_id}/stats/devices` - Site-specific device statistics
- `GET /api/v1/sites/{site_id}/discovered_switches` - **Critical for unmanaged switch discovery**
- `GET /api/v1/sites/{site_id}/devices/{device_id}/ports` - Individual port information

Network insights and connectivity endpoints

Advanced topology analysis through insights and SLE metrics:

- `GET /api/v1/orgs/{org_id}/insights/{metric}` - Organization-wide topology insights
- `GET /api/v1/orgs/{org_id}/sle/summary` - Service Level Experience metrics across all sites

Authentication requirements and API call structure

Token-based authentication provides the most secure and efficient access method for programmatic topology retrieval. Mist supports user tokens (bound to individual accounts) and organization tokens (independent of specific users).

API token generation and management

Creating API tokens through the Mist portal:

1. Log into your Mist portal and navigate to account settings
2. Select "API Token" section and click "Create Token"

3. **Critical security note**: Copy the full token immediately - it will never be displayed again
4. Store tokens securely using environment variables rather than hardcoded values

Organization tokens provide superior scalability with individual rate limiting (5,000 calls per hour per token) compared to user tokens that share limits across all tokens for the same user account.

Required HTTP headers and authentication format

Standard API call structure:

```
curl -H "Content-Type: application/json" \
     -H "Authorization: Token YOUR_API_TOKEN_HERE" \
     -X GET https://api.mist.com/api/v1/ENDPOINT
```

Essential headers for all requests:

- `Authorization: Token <your-api-token>` (never use Basic auth or session cookies for programmatic access)
- `Content-Type: application/json` (required for all Mist API interactions)

Rate limiting optimization: With bulk endpoints, you can retrieve an entire organization's topology in 2-3 API calls instead of hundreds, staying well within the 5,000 calls/hour limit.

Efficient bulk retrieval implementation

Complete Python implementation using organization-level endpoints

```python
import json
import requests
import time
from typing import Dict, List, Optional

class MistBulkTopologyClient:
    def __init__(self, token: str, org_id: str, host: str = "api.mist.com"):
        self.token = token
        self.org_id = org_id
        self.base_url = f"https://{host}/api/v1"
        self.headers = {
            'Content-Type': 'application/json',
            'Authorization': f'Token {token}'
        }
        self.topology_cache = {}

    def get_complete_topology(self) -> Dict:
        """
        Retrieve complete non-EVPN topology for entire organization
        using minimal API calls (2-3 total)
        """
        print("Fetching complete organization topology...")

        # Step 1: Single call for ALL devices across ALL sites
        all_devices = self._get_organization_inventory()

        # Step 2: Single call for ALL device statistics
        all_stats = self._get_organization_stats()

        # Step 3: Build complete topology map locally
        topology = self._build_topology_map(all_devices, all_stats)

        # Optional Step 4: Get discovered switches (requires site iteration)
        # Only if unmanaged device discovery is needed
        if self._needs_discovered_switches():
            topology['discovered_switches'] =
self._get_discovered_switches_bulk(topology['sites'])

        return topology

    def _get_organization_inventory(self) -> List[Dict]:
        """Get all devices across organization in single API call"""
        url = f"{self.base_url}/orgs/{self.org_id}/inventory"
        print(f"API Call 1: Getting organization inventory...")
        return self._make_request(url) or []

    def _get_organization_stats(self) -> List[Dict]:
        """Get all device statistics in single API call"""
        url = f"{self.base_url}/orgs/{self.org_id}/stats/devices"
        print(f"API Call 2: Getting organization-wide device statistics...")
        return self._make_request(url) or []

    def _build_topology_map(self, devices: List[Dict], stats: List[Dict]) -> Dict:
        """
        Process bulk data locally to build complete topology map
        without additional API calls
        """
        # Create lookup tables for efficient processing
        stats_by_mac = {stat['mac']: stat for stat in stats if 'mac' in stat}

        topology = {
            'organization_id': self.org_id,
            'total_devices': len(devices),
            'sites': {},
            'devices_by_type': {'switch': [], 'ap': [], 'gateway': []},
            'topology_links': [],
            'device_connections': {}
        }

        # Process each device from inventory
        for device in devices:
            site_id = device.get('site_id', 'unassigned')
            device_type = device.get('type', 'unknown')
            device_mac = device.get('mac')

            # Initialize site if not exists
            if site_id not in topology['sites']:
                topology['sites'][site_id] = {
                    'site_id': site_id,
                    'site_name': device.get('site_name', 'Unknown'),
                    'devices': [],
                    'device_count': 0
                }

            # Build device entry with stats if available
            device_entry = {
                'name': device.get('name'),
                'mac': device_mac,
                'serial': device.get('serial'),
                'model': device.get('model'),
                'type': device_type,
                'site_id': site_id
            }

            # Merge statistics if available
            if device_mac in stats_by_mac:
```

```python
                device_stats = stats_by_mac[device_mac]
                device_entry['status'] = device_stats.get('status', 'unknown')
                device_entry['uptime'] = device_stats.get('uptime')
                device_entry['version'] = device_stats.get('version')

                # Extract connectivity information
                connections = self._extract_connections_from_stats(device_stats)
                if connections:
                    device_entry['connections'] = connections
                    topology['device_connections'][device_mac] = connections

                    # Build topology links
                    for conn in connections:
                        topology['topology_links'].append({
                            'source_mac': device_mac,
                            'source_port': conn['port'],
                            'source_name': device.get('name'),
                            'target_mac': conn.get('neighbor_mac'),
                            'target_port': conn.get('neighbor_port'),
                            'link_status': conn.get('status', 'up'),
                            'speed_mbps': conn.get('speed'),
                            'protocol': conn.get('protocol', 'LLDP')
                        })

            # Add to appropriate collections
            topology['sites'][site_id]['devices'].append(device_entry)
            topology['sites'][site_id]['device_count'] += 1

            if device_type in topology['devices_by_type']:
                topology['devices_by_type'][device_type].append(device_entry)

        # Calculate topology statistics
        topology['statistics'] = self._calculate_topology_stats(topology)

        return topology

    def _extract_connections_from_stats(self, device_stats: Dict) -> List[Dict]:
        """Extract all connectivity information from device statistics"""
        connections = []

        # Process port statistics
        if 'port_stat' in device_stats:
            for port in device_stats['port_stat']:
                if port.get('up'):
                    connection = {
                        'port': port.get('port_id'),
                        'status': 'up',
                        'speed': port.get('speed'),
                        'rx_bytes': port.get('rx_bytes', 0),
                        'tx_bytes': port.get('tx_bytes', 0)
                    }

                    # Add neighbor information if available
                    if port.get('neighbor_mac'):
                        connection.update({
                            'neighbor_mac': port['neighbor_mac'],
                            'neighbor_port': port.get('neighbor_port'),
                            'neighbor_system': port.get('neighbor_system_name')
                        })

                    connections.append(connection)
        # Process LLDP information
        if 'lldp_stat' in device_stats:
            for lldp in device_stats['lldp_stat']:
                connection = {
                    'port': lldp.get('local_port'),
                    'neighbor_mac': lldp.get('chassis_id'),
                    'neighbor_port': lldp.get('port_id'),
                    'neighbor_system': lldp.get('system_name'),
                    'protocol': 'LLDP',
                    'status': 'discovered'
                }
                connections.append(connection)

        return connections

    def _calculate_topology_stats(self, topology: Dict) -> Dict:
        """Calculate topology statistics from processed data"""
        total_links = len(topology['topology_links'])
        unique_links = len(set(
            (min(link['source_mac'], link.get('target_mac', '')),
             max(link['source_mac'], link.get('target_mac', '')))
            for link in topology['topology_links']
            if link.get('target_mac')
        ))

        return {
            'total_sites': len(topology['sites']),
            'total_devices': topology['total_devices'],
            'total_switches': len(topology['devices_by_type']['switch']),
            'total_aps': len(topology['devices_by_type']['ap']),
            'total_gateways': len(topology['devices_by_type']['gateway']),
            'total_connections': total_links,
            'unique_links': unique_links,
            'devices_with_connections': len(topology['device_connections'])
        }

    def _needs_discovered_switches(self) -> bool:
        """Determine if discovered switches retrieval is needed"""
        # Implement your logic here - for example:
        # return True if you need to discover unmanaged switches
        return False

    def _get_discovered_switches_bulk(self, sites: Dict) -> Dict:
        """
        Get discovered switches for all sites (requires site-level calls)
        Only use when unmanaged device discovery is essential
        """
        discovered = {}
        for site_id in sites.keys():
            url = f"{self.base_url}/sites/{site_id}/discovered_switches"
            print(f"Additional API Call: Getting discovered switches for site {site_id}")
            result = self._make_request(url)
            if result:
                discovered[site_id] = result
        return discovered

    def _make_request(self, url: str) -> Optional[Dict]:
        """Make API request with error handling and rate limiting"""
        try:
            response = requests.get(url, headers=self.headers)

            # Handle rate limiting
            if response.status_code == 429:
                print("Rate limit reached, waiting 60 seconds...")
                time.sleep(60)
                response = requests.get(url, headers=self.headers)

            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            print(f"Error making request to {url}: {e}")
            return None

    def export_topology_to_file(self, topology: Dict, filename: str = "topology.json"):
        """Export topology to JSON file"""
        with open(filename, 'w') as f:
            json.dump(topology, f, indent=2)
        print(f"Topology exported to {filename}")

# Usage example
if __name__ == "__main__":
    # Initialize client
    client = MistBulkTopologyClient(
        token="your-api-token-here",
        org_id="your-org-id-here"
    )

    # Get complete topology with just 2 API calls
    topology = client.get_complete_topology()

    # Display statistics
    stats = topology.get('statistics', {})
    print(f"\n=== Topology Discovery Complete ===")
    print(f"Total API Calls Made: 2")
    print(f"Sites: {stats.get('total_sites', 0)}")
    print(f"Devices: {stats.get('total_devices', 0)}")
    print(f"Switches: {stats.get('total_switches', 0)}")
    print(f"Unique Network Links: {stats.get('unique_links', 0)}")

    # Export to file
    client.export_topology_to_file(topology)
```

Essential curl commands for bulk topology retrieval

Get ALL devices across entire organization (single call):

```bash
curl -H "Authorization: Token YOUR_TOKEN" \
     https://api.mist.com/api/v1/orgs/{org_id}/inventory
```

Get organization-wide device statistics (single call):

```bash
curl -H "Authorization: Token YOUR_TOKEN" \
     https://api.mist.com/api/v1/orgs/{org_id}/stats/devices
```

Search for specific device types across organization:

```bash
curl -H "Authorization: Token YOUR_TOKEN" \
     "https://api.mist.com/api/v1/orgs/{org_id}/devices/search?type=switch&limit=1000"
```

Export complete device configurations:

```
curl -H "Authorization: Token YOUR_TOKEN" \
     https://api.mist.com/api/v1/orgs/{org_id}/devices/export
```

Comparison: Inefficient vs Efficient Approach

❌ Inefficient Method (Avoid This)

```python
# Makes potentially hundreds of API calls
def get_topology_inefficient(org_id):
    sites = get_all_sites(org_id)  # 1 API call
    topology = {}

    for site in sites:  # For 50 sites...
        devices = get_site_devices(site['id'])  # 50 API calls

        for device in devices:  # For 20 devices per site...
            device_detail = get_device_detail(device['id'])  # 1000 API calls
            device_stats = get_device_stats(device['id'])  # 1000 more API calls

    # Total: 2051+ API calls
    return topology
```

✅ Efficient Method (Use This)

```python
# Makes only 2 API calls
def get_topology_efficient(org_id):
    all_devices = get_org_inventory(org_id)  # 1 API call
    all_stats = get_org_device_stats(org_id)  # 1 API call

    # Process everything locally
    topology = build_topology_from_bulk_data(all_devices, all_stats)

    # Total: 2 API calls
    return topology
```

Key differences between EVPN and non-EVPN topology retrieval

EVPN topologies use fundamentally different API endpoints and data structures compared to traditional network discovery methods. Understanding these distinctions ensures proper implementation approach selection.

Architectural and endpoint differences

EVPN topology management operates through dedicated campus fabric endpoints:

- `GET /api/v1/orgs/{org_id}/evpn_topologies` - EVPN fabric configuration and status
- Requires BGP AS numbers, overlay/underlay network definitions, and VXLAN parameters
- **Provides centralized fabric-wide visibility** with native traffic isolation through VRFs

Traditional topology discovery relies on device-level endpoints and protocols:

- Uses organization-wide inventory and statistics endpoints for bulk retrieval
- **Limited to Layer 2/3 connectivity without overlay abstractions**
- Requires manual VLAN planning and Spanning Tree Protocol for loop prevention

Data structure and response format differences

EVPN responses include fabric-specific metadata:

```
{
  "overlay": {
    "as": 65000,
    "name": "campus-fabric"
  },
  "underlay": {
    "subnet": "10.0.0.0/16",
    "routed_at": "core"
  },
  "switches": [
    {
      "role": "core",
      "mac": "device-mac",
      "vtep_ip": "10.0.1.1"
    }
  ]
}
```

Non-EVPN bulk responses focus on physical connectivity:

```
{
  "inventory": [
    {
      "mac": "device-mac-1",
      "name": "switch-01",
      "site_id": "site-uuid",
      "type": "switch",
      "model": "EX4400-48P"
    }
  ],
  "stats": [
    {
      "mac": "device-mac-1",
      "port_stat": [
        {
          "port_id": "ge-0/0/1",
          "up": true,
          "neighbor_mac": "connected-device-mac"
        }
      ]
    }
  ]
}
```

Configuration requirements for non-EVPN visibility

Essential switch configuration for topology discovery:

```
set protocols lldp interface all
set protocols lldp management-address x.x.x.x
set protocols lldp port-id-subtype interface-name
```

LLDP enablement is critical - without proper LLDP configuration, the discovered switches endpoint will not populate with unmanaged device information.

Implementation best practices and optimization strategies

Rate Limiting Optimization

Bulk retrieval strategy benefits:

- **2-3 API calls** for complete organization topology vs hundreds with individual queries
- **5,000 calls/hour limit** becomes a non-issue with bulk endpoints
- **Consistent data snapshot** - all information retrieved at the same point in time
- **Reduced latency** - especially important for geographically distributed deployments

Performance Optimization Techniques

1. **Implement local caching**: Store bulk data locally and refresh periodically
2. **Use pagination wisely**: Organization endpoints support up to 1000 items per page
3. **Filter at the API level**: Use query parameters to reduce payload size
4. **Process data asynchronously**: Parse bulk responses in parallel threads/processes

Error Handling Best Practices

```python
def robust_api_call(url, headers, max_retries=3):
    for attempt in range(max_retries):
        try:
            response = requests.get(url, headers=headers, timeout=30)
            if response.status_code == 429:
                wait_time = int(response.headers.get('Retry-After', 60))
                time.sleep(wait_time)
                continue
            response.raise_for_status()
            return response.json()
        except requests.exceptions.Timeout:
            if attempt == max_retries - 1:
                raise
            time.sleep(2 ** attempt)  # Exponential backoff
    return None
```

Data Freshness Considerations

- **Device statistics** update every 2-5 minutes
- **Inventory data** reflects real-time device presence
- **LLDP information** refreshes based on protocol timers (typically 30 seconds)
- **For real-time updates**: Implement webhook subscriptions instead of polling

Official documentation references

Primary API documentation: https://www.juniper.net/documentation/us/en/software/mist/api/

Interactive API testing: https://api.mist.com/api/v1/docs (requires authentication)

GitHub resources: https://github.com/tmunzer/mist_library (production Python examples)

Postman collections: https://www.postman.com/juniper-mist/workspace/mist-systems-s-public-workspace

These resources provide comprehensive implementation guidance with working examples optimized for bulk retrieval and efficient topology discovery in non-EVPN environments.