TR181 Node Comparator Troubleshooting Guide

Table of Contents

Common Connection Issues

Issue: Connection Timeout

Symptoms: - Operations hang and eventually timeout - Error message: “Connection timeout after X seconds”

Possible Causes: - Network connectivity issues - Device is unreachable - Firewall blocking connections - Device is overloaded or unresponsive

Solutions:

1. **Check network connectivity:**

   ```
   ping device.local
   telnet device.local 7547  # For CWMP
   curl -I http://device.local/api  # For REST API
   ```

2. **Increase timeout values:**

   ```
   device_config = DeviceConfig(
       name="slow_device",
       type="cwmp",
       endpoint="http://device.local:7547/cwmp",
       timeout=120,  # Increase from default 30 seconds
       retry_count=5
   )
   ```

3. **Check firewall settings:**

   ```
   # Check if ports are open
   nmap -p 7547 device.local  # CWMP
   nmap -p 80,443 device.local  # HTTP/HTTPS
   ```

4. **Verify device status:**

   - Check device logs for errors
   - Verify device is not in maintenance mode

- Ensure device has sufficient resources

Issue: Connection Refused

Symptoms: - Immediate connection failure - Error message: “Connection refused” or “No route to host”

Possible Causes: - Wrong IP address or hostname - Service not running on target port - Network routing issues

Solutions:

1. **Verify endpoint configuration:**

```
# Check if endpoint is correct
import requests
try:
    response = requests.get("http://device.local/api", timeout=5)
    print(f"Status: {response.status_code}")
except requests.exceptions.ConnectionError:
    print("Connection refused - check endpoint")
```

2. **Check service status on device:**

```
# SSH to device and check services
systemctl status cwmp-service  # For CWMP
systemctl status api-service   # For REST API
```

3. **Verify port configuration:**

```
# Common CWMP ports: 7547, 8080, 80
# Common API ports: 80, 443, 8080, 3000
endpoints_to_try = [
    "http://device.local:7547/cwmp",
    "http://device.local:8080/cwmp",
    "http://device.local/cwmp"
]
```

Issue: SSL/TLS Certificate Errors

Symptoms: - SSL verification failures - Certificate validation errors - “SSL: CERTIFICATE_VERIFY_FAILED” errors

Solutions:

1. **Disable SSL verification (development only):**

```
device_config = DeviceConfig(
    name="dev_device",
    type="rest",
    endpoint="https://device.local/api",
    hook_config=HookConfig(
        ssl_verify=False  # Only for development!
    )
)
```

2. **Provide custom CA certificate:**

```
device_config = DeviceConfig(
    name="secure_device",
```

```
    type="rest",
    endpoint="https://device.local/api",
    hook_config=HookConfig(
        ssl_verify=True,
        ssl_ca_path="/path/to/ca-bundle.crt"
    )
)
```

3. **Use client certificates:**

```
device_config = DeviceConfig(
    name="client_cert_device",
    type="cwmp",
    endpoint="https://device.local:7548/cwmp",
    hook_config=HookConfig(
        ssl_cert_path="/path/to/client.crt",
        ssl_key_path="/path/to/client.key"
    )
)
```

Authentication Problems

Issue: Invalid Credentials

Symptoms: - HTTP 401 Unauthorized errors - Authentication failure messages - Access denied errors

Solutions:

1. **Verify credentials:**

```
# Test credentials manually
import requests
from requests.auth import HTTPBasicAuth

response = requests.get(
    "http://device.local/api",
    auth=HTTPBasicAuth('username', 'password')
)
print(f"Status: {response.status_code}")
```

2. **Check authentication type:**

```
# Try different authentication methods
configs = [
    {"type": "basic", "username": "admin", "password": "password"},
    {"type": "digest", "username": "admin", "password": "password"},
    {"type": "bearer", "token": "your-token"},
    {"type": "api_key", "key": "your-key", "header": "X-API-Key"}
]
```

3. **Handle password special characters:**

```
import urllib.parse

# URL encode passwords with special characters
password = "p@ssw0rd!#$"
```

```
encoded_password = urllib.parse.quote(password)
```

Issue: Token Expiration

Symptoms: - Authentication works initially but fails later - “Token expired” or “Invalid token” errors - HTTP 401 errors after successful operations

Solutions:

1. **Implement token refresh:**

```
device_config = DeviceConfig(
    name="oauth_device",
    type="rest",
    endpoint="https://device.local/api",
    authentication={
        "type": "oauth2",
        "client_id": "your-client-id",
        "client_secret": "your-client-secret",
        "token_url": "https://auth.example.com/token"
    },
    hook_config=HookConfig(
        oauth_refresh_threshold=300  # Refresh 5 minutes before expiry
    )
)
```

2. **Check token validity:**

```
import jwt
import time

def is_token_expired(token):
    try:
        decoded = jwt.decode(token, options={"verify_signature": False})
        exp = decoded.get('exp', 0)
        return time.time() > exp
    except:
        return True
```

Data Validation Errors

Issue: Invalid TR181 Path Format

Symptoms: - ValidationError: “Invalid TR181 path format” - Path validation failures

Solutions:

1. **Check path format:**

```
# Valid TR181 paths
valid_paths = [
    "Device.WiFi.Radio.1.Channel",
    "Device.DeviceInfo.Manufacturer",
    "Device.Ethernet.Interface.1.Enable"
]

# Invalid paths
```

```
invalid_paths = [
    "device.wifi.radio.1.channel",  # Wrong case
    "WiFi.Radio.1.Channel",         # Missing Device prefix
    "Device.WiFi..Radio.1.Channel", # Double dots
    "Device.WiFi.Radio.1.",         # Trailing dot
]
```

2. **Validate paths before processing:**

```
import re

def validate_tr181_path(path):
    # TR181 path pattern
    pattern = r'^Device\.([A-Z][a-zA-Z0-9]*\.)*[A-Z][a-zA-Z0-9]*$'
    return re.match(pattern, path) is not None

# Usage
if not validate_tr181_path(node.path):
    print(f"Invalid path: {node.path}")
```

Issue: Data Type Mismatches

Symptoms: - ValidationError: “Expected int, got string” - Type validation failures

Solutions:

1. **Check data type mapping:**

```
# Common TR181 data types and their Python equivalents
type_mapping = {
    'string': str,
    'int': int,
    'unsignedInt': int,
    'boolean': bool,
    'dateTime': str,  # ISO format string
    'base64': str,
    'hexBinary': str
}
```

2. **Implement type conversion:**

```
def convert_value(value, expected_type):
    if expected_type == 'int':
        return int(value) if value is not None else None
    elif expected_type == 'boolean':
        if isinstance(value, str):
            return value.lower() in ('true', '1', 'yes', 'on')
        return bool(value)
    elif expected_type == 'string':
        return str(value) if value is not None else None
    return value
```

Issue: Value Range Violations

Symptoms: - ValidationError: “Value X exceeds maximum Y” - Range validation failures

Solutions:

1. **Check value constraints:**

```
from tr181_comparator import ValueRange

# Define proper value ranges
channel_range = ValueRange(
    min_value=1,
    max_value=11,
    allowed_values=[1, 6, 11]  # Common non-overlapping channels
)

ssid_range = ValueRange(
    max_length=32,
    pattern=r'^[a-zA-Z0-9_-]+$'  # Alphanumeric, underscore, hyphen
)
```

2. **Validate values before assignment:**

```
def validate_value_range(value, range_spec):
    if range_spec.allowed_values and value not in range_spec.allowed_values:
        raise ValueError(f"Value {value} not in allowed values")

    if range_spec.min_value is not None and value < range_spec.min_value:
        raise ValueError(f"Value {value} below minimum {range_spec.min_value}")

    if range_spec.max_value is not None and value > range_spec.max_value:
        raise ValueError(f"Value {value} above maximum {range_spec.max_value}")
```

Performance Issues

Issue: Slow Extraction Operations

Symptoms: - Long delays during node extraction - High memory usage - Timeouts on large datasets

Solutions:

1. **Use batch processing:**

```
async def batch_extraction(extractor, batch_size=100):
    all_nodes = []
    paths = await extractor.get_parameter_names()

    for i in range(0, len(paths), batch_size):
        batch_paths = paths[i:i + batch_size]
        batch_nodes = await extractor.extract_batch(batch_paths)
        all_nodes.extend(batch_nodes)

        # Optional: Add delay between batches
        await asyncio.sleep(0.1)

    return all_nodes
```

2. **Implement connection pooling:**

```
device_config = DeviceConfig(
    name="high_performance_device",
    type="rest",
    endpoint="http://device.local/api",
    hook_config=HookConfig(
        connection_pool_size=10,
        keep_alive=True,
        max_connections_per_host=5
    )
)
```

3. **Use filtering to reduce data:**

```
# Extract only specific parameter subtrees
filtered_extractor = CWMPExtractor(config)
wifi_nodes = await filtered_extractor.extract_subtree("Device.WiFi")
device_info = await filtered_extractor.extract_subtree("Device.DeviceInfo")
```

Issue: Memory Usage Problems

Symptoms: - Out of memory errors - System becomes unresponsive - Memory usage grows continuously

Solutions:

1. **Process data in chunks:**

```
def process_large_comparison(nodes1, nodes2, chunk_size=1000):
    results = []

    for i in range(0, len(nodes1), chunk_size):
        chunk1 = nodes1[i:i + chunk_size]
        chunk2 = nodes2[i:i + chunk_size]

        chunk_result = compare_chunk(chunk1, chunk2)
        results.append(chunk_result)

        # Clear references to help garbage collection
        del chunk1, chunk2

    return combine_results(results)
```

2. **Use generators for large datasets:**

```
def node_generator(extractor):
    """Generator that yields nodes one at a time"""
    paths = extractor.get_parameter_names()
    for path in paths:
        yield extractor.get_node(path)

# Usage
for node in node_generator(extractor):
    process_node(node)
```

Configuration Problems

Issue: Invalid Configuration Format

Symptoms: - JSON/YAML parsing errors - Configuration validation failures - Missing required fields

Solutions:

1. **Validate JSON syntax:**

```
# Use jq to validate JSON
jq . config.json

# Use Python to validate
python -m json.tool config.json
```

2. **Validate YAML syntax:**

```
# Use yamllint
yamllint config.yaml

# Use Python to validate
python -c "import yaml; yaml.safe_load(open('config.yaml'))"
```

3. **Use configuration schema validation:**

```
import jsonschema

config_schema = {
    "type": "object",
    "required": ["devices", "subsets"],
    "properties": {
        "devices": {
            "type": "array",
            "items": {
                "type": "object",
                "required": ["name", "type", "endpoint"],
                "properties": {
                    "name": {"type": "string"},
                    "type": {"type": "string", "enum": ["cwmp", "rest", "snmp"]},
                    "endpoint": {"type": "string"}
                }
            }
        }
    }
}

# Validate configuration
try:
    jsonschema.validate(config_data, config_schema)
except jsonschema.ValidationError as e:
    print(f"Configuration error: {e.message}")
```

Issue: Environment Variable Substitution

Symptoms: - Environment variables not resolved - Literal "${VAR}" strings in configuration - Authentication failures due to unresolved variables

Solutions:

1. **Implement environment variable substitution:**

```python
import os
import re

def substitute_env_vars(text):
    def replace_var(match):
        var_name = match.group(1)
        default_value = match.group(3) if match.group(3) else ""
        return os.environ.get(var_name, default_value)

    # Pattern: ${VAR} or ${VAR:default}
    pattern = r'\$\{([^}:]+)(:([^}]*))?\}'
    return re.sub(pattern, replace_var, text)

# Usage
config_text = '{"password": "${DB_PASSWORD:default_pass}"}'
resolved_config = substitute_env_vars(config_text)
```

2. **Use environment file:**

```
# .env file
DEVICE_PASSWORD=secure_password_123
API_TOKEN=your_api_token_here

# Load in Python
from dotenv import load_dotenv
load_dotenv()
```

CWMP-Specific Issues

Issue: SOAP Envelope Errors

Symptoms: - SOAP parsing errors - Malformed envelope messages - Protocol version mismatches

Solutions:

1. **Check SOAP version compatibility:**

```
device_config = DeviceConfig(
    name="cwmp_device",
    type="cwmp",
    endpoint="http://device.local:7547/cwmp",
    hook_config=HookConfig(
        soap_version="1.1",  # or "1.2"
        max_envelope_size=65536
    )
)
```

2. **Enable SOAP debugging:**

```
import logging
logging.getLogger('soap').setLevel(logging.DEBUG)
```

Issue: RPC Method Not Supported

Symptoms: - “Method not supported” errors - Missing RPC operations - Limited device functionality

Solutions:

1. **Check supported methods:**

```
async def check_supported_methods(cwmp_extractor):
    try:
        methods = await cwmp_extractor.get_rpc_methods()
        print("Supported methods:", methods)
    except Exception as e:
        print(f"Cannot get RPC methods: {e}")
```

2. **Use alternative methods:**

```
# If GetParameterNames not supported, try GetParameterValues with known paths
known_paths = [
    "Device.DeviceInfo.",
    "Device.WiFi.",
    "Device.Ethernet."
]

for path in known_paths:
    try:
        values = await extractor.get_parameter_values([path])
    except Exception:
        continue
```

REST API Issues

Issue: API Rate Limiting

Symptoms: - HTTP 429 “Too Many Requests” errors - Temporary API blocks - Degraded performance

Solutions:

1. **Implement rate limiting:**

```
import asyncio
from asyncio import Semaphore

class RateLimitedHook(RESTAPIHook):
    def __init__(self, requests_per_second=10):
        super().__init__()
        self.semaphore = Semaphore(requests_per_second)
        self.last_request_time = 0
        self.min_interval = 1.0 / requests_per_second

    async def make_request(self, *args, **kwargs):
        async with self.semaphore:
```

```
            now = time.time()
            time_since_last = now - self.last_request_time
            if time_since_last < self.min_interval:
                await asyncio.sleep(self.min_interval - time_since_last)

            self.last_request_time = time.time()
            return await super().make_request(*args, **kwargs)
```

2. **Handle rate limit responses:**

```
async def handle_rate_limit(response):
    if response.status_code == 429:
        retry_after = response.headers.get('Retry-After', '60')
        wait_time = int(retry_after)
        print(f"Rate limited. Waiting {wait_time} seconds...")
        await asyncio.sleep(wait_time)
        return True  # Retry
    return False  # Don't retry
```

Issue: API Version Compatibility

Symptoms: - "API version not supported" errors - Unexpected response formats - Missing endpoints

Solutions:

1. **Check API version:**

```
async def check_api_version(endpoint):
    try:
        response = await session.get(f"{endpoint}/version")
        version_info = await response.json()
        print(f"API Version: {version_info}")
        return version_info
    except Exception as e:
        print(f"Cannot determine API version: {e}")
```

2. **Use version-specific endpoints:**

```
device_config = DeviceConfig(
    name="versioned_api_device",
    type="rest",
    endpoint="http://device.local/api/v2",  # Specify version
    hook_config=HookConfig(
        api_version="v2",
        version_header="X-API-Version"
    )
)
```

Comparison and Validation Issues

Issue: False Positive Differences

Symptoms: - Many differences reported for identical values - String/numeric comparison issues - Case sensitivity problems

Solutions:

1. **Normalize values before comparison:**

```
def normalize_value(value, data_type):
    if data_type == 'boolean':
        if isinstance(value, str):
            return value.lower() in ('true', '1', 'yes', 'on')
        return bool(value)
    elif data_type == 'int':
        return int(value) if value is not None else None
    elif data_type == 'string':
        return str(value).strip() if value is not None else None
    return value
```

2. **Use fuzzy comparison for strings:**

```
from difflib import SequenceMatcher

def fuzzy_string_compare(str1, str2, threshold=0.9):
    similarity = SequenceMatcher(None, str1, str2).ratio()
    return similarity >= threshold
```

Issue: Missing Node Relationships

Symptoms: - Orphaned child nodes - Broken parent-child relationships - Incomplete hierarchical structure

Solutions:

1. **Build relationships after extraction:**

```
def build_node_relationships(nodes):
    node_map = {node.path: node for node in nodes}

    for node in nodes:
        # Find parent
        parent_path = '.'.join(node.path.split('.')[:-1])
        if parent_path in node_map:
            node.parent = parent_path
            if not node_map[parent_path].children:
                node_map[parent_path].children = []
            node_map[parent_path].children.append(node.path)
```

2. **Validate node hierarchy:**

```
def validate_hierarchy(nodes):
    issues = []
    node_paths = {node.path for node in nodes}

    for node in nodes:
        if node.parent and node.parent not in node_paths:
            issues.append(f"Missing parent {node.parent} for {node.path}")

        if node.children:
            for child_path in node.children:
                if child_path not in node_paths:
```

```
                    issues.append(f"Missing child {child_path} for {node.path}")

        return issues
```

Logging and Debugging

Enable Debug Logging

```
import logging

# Configure logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('tr181_debug.log'),
        logging.StreamHandler()
    ]
)

# Enable specific loggers
logging.getLogger('tr181_comparator').setLevel(logging.DEBUG)
logging.getLogger('requests').setLevel(logging.DEBUG)
logging.getLogger('urllib3').setLevel(logging.DEBUG)
```

Capture Network Traffic

```
# Use tcpdump to capture network traffic
sudo tcpdump -i any -w tr181_traffic.pcap host device.local

# Use Wireshark to analyze captured traffic
wireshark tr181_traffic.pcap
```

Debug CWMP Communication

```
# Enable SOAP message logging
import logging
logging.getLogger('zeep.transports').setLevel(logging.DEBUG)

# Custom SOAP logging
class DebugCWMPHook(CWMPHook):
    async def send_soap_request(self, request):
        print(f"SOAP Request: {request}")
        response = await super().send_soap_request(request)
        print(f"SOAP Response: {response}")
        return response
```

Error Reference

ConnectionError

Description: Raised when unable to establish connection to device

Common Causes: - Network connectivity issues - Wrong endpoint configuration - Authentication failures - SSL/TLS certificate problems

Resolution Steps: 1. Check network connectivity 2. Verify endpoint URL and port 3. Test authentication credentials 4. Check SSL certificate configuration

ValidationError

Description: Raised when data validation fails

Common Causes: - Invalid TR181 path format - Data type mismatches - Value range violations - Missing required fields

Resolution Steps: 1. Validate TR181 path format 2. Check data type mappings 3. Verify value constraints 4. Review node definitions

ConfigurationError

Description: Raised when configuration is invalid

Common Causes: - Invalid JSON/YAML syntax - Missing required configuration fields - Invalid configuration values - Environment variable resolution failures

Resolution Steps: 1. Validate configuration file syntax 2. Check required fields are present 3. Verify configuration values 4. Test environment variable substitution

TimeoutError

Description: Raised when operations exceed timeout limits

Common Causes: - Slow network connections - Overloaded devices - Large dataset processing - Insufficient timeout values

Resolution Steps: 1. Increase timeout values 2. Check network performance 3. Verify device performance 4. Use batch processing for large datasets

AuthenticationError

Description: Raised when authentication fails

Common Causes: - Invalid credentials - Expired tokens - Wrong authentication method - Missing authentication headers

Resolution Steps: 1. Verify credentials are correct 2. Check token expiration 3. Confirm authentication method 4. Review authentication headers

Getting Help

If you continue to experience issues:

1. **Check the logs:** Enable debug logging to get detailed error information
2. **Review configuration:** Validate all configuration files and settings
3. **Test connectivity:** Verify network connectivity and device accessibility
4. **Consult documentation:** Review API documentation and user guide
5. **Report issues:** Create detailed bug reports with logs and configuration

Creating Effective Bug Reports

Include the following information:

1. **Environment details:**
   - Python version
   - Operating system
   - Network configuration
2. **Configuration:**
   - Device configuration (sanitized)
   - System configuration
   - Command line arguments
3. **Error details:**
   - Complete error messages
   - Stack traces
   - Debug logs
4. **Reproduction steps:**
   - Exact steps to reproduce the issue
   - Expected vs actual behavior
   - Minimal test case if possible