16-12-2020

Zero-Footprint Monitoring

Grant Agreement No.:	856726
Work Package	WP6
Task Item:	Task 3
Dissemination Level:	PU (Public)
Document ID:	GN4-3-20-2727118

Authors: Pavle Vuletić (University of Belgrade), Marinos Dimolianis (NTUA/GRNET), Victor Olifer (Jisc), Ivana Golub (PSNC)

© GÉANT Association on behalf of the GN4-3 project.
The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 856726 (GN4-3).

Abstract
Zero-footprint monitoring is monitoring performed without the use of additional dedicated hardware elements. This document describes a set of new network element features that enable scalable and vendor-independent network service performance verification. It is written in a cookbook style and describes the use of standard protocols on network elements and virtual services on top of network elements, as well as the use of streaming telemetry for result transportation towards result repositories.

Table of Contents

Table of Figures

Table of Tables

Executive Summary

For a long time, gathering network service performance parameters in multi-vendor environments was a challenge, requiring either the use of dedicated hardware probes, which increased the hardware footprint and the complexity of an organisation's network deployment, or the use of proprietary, vendor-specific and non-interoperable features on network elements.

More recently, some standard protocols for network performance evaluation, such as the Two-Way Active Measurement Protocol (TWAMP), have started to appear as a regular feature in network elements of different vendors. This, together with the emerging virtualisation trend, has enabled the use of standard-based protocols for performance evaluation across a range of different devices of various vendors without the use of additional equipment, in a model known as zero-footprint monitoring.

This document describes the current state of the art of standard network service performance monitoring protocols and how they can be configured on different platforms (Juniper, Cisco, Linux) as well as on virtual services on top of Cisco routers, which are an interesting recent development. It is written in a cookbook style which allows the reader to quickly test all the proposed monitoring methods and use them in operational environments.

Besides this standards-based approach to network service performance monitoring, guidelines are provided for transferring the results of the measurements using the emerging streaming telemetry approach. As it is expected that streaming telemetry will soon replace the Simple Network Management Protocol (SNMP) as a method for gathering network monitoring data, this document also aims to be a useful resource for the early adopters of this technology.

1 Introduction

Monitoring and performance verification are essential groups of processes in network service operations. These processes allow providers to verify the health of the products and services they are offering and the operation of the underlying infrastructure, and to prove the quality of the services (QoS) delivered to their users. Users might also wish to have insight into the performance of the services they use, especially in cases where strict QoS requirements apply.

Such service-oriented performance views are increasingly important, especially given that modern network services are commonly virtualised, with the traffic of multiple users or customers multiplexed over the same physical links. With this type of service, it is impossible to estimate the quality of users' experience or to verify the Service Level Agreement (SLA) by using legacy tools which only monitor physical links and network element operation. A situation can easily be envisaged where the physical link operates without any flaw or congestion, while on the other hand one or multiple users do not get proper service over that link due to a virtual network misconfiguration or some problem in the network virtualisation software stack. Therefore, new monitoring methods for performance verification within a network service are required.

Verifying network service performance requires gathering well-known service performance indicators such as packet latency, latency variation (jitter) and/or packet loss rate between the service endpoints and comparing them to threshold values, or against historical data. Ten years ago, network performance monitoring often assumed the use of dedicated monitoring equipment in network PoPs and data centres. Monitoring probes, databases to hold results, and UI servers would have to be installed. Monitoring network service performance from each PoP meant the use of a separate monitoring probe/machine in each of these. Some examples of such architectures include the discontinued project RIPE TTM or perfSONAR versions developed before the GN4-1 project[1].

In recent years there have been several attempts to make the monitoring process more lightweight and automated, but also to decrease the size of the monitoring agents (e.g., the RIPE ATLAS project or the "small node" PMP service in the GÉANT and European NREN networks). With the emergence of virtualisation, it is now also possible to install some of the most popular tools on virtual machines or even as applications on containers, which may reside on network hardware.

The decrease in monitoring probe size comes at a cost. The use of virtual appliances or lower-end hardware typically meant that capacity and available bandwidth measurements were of limited accuracy, especially on high-speed links (above 1Gbps). However, this does not necessarily present a problem, as bandwidth measurements must saturate a path with traffic in order to provide accurate capacity measurement results where available. Such measurements are intrusive and disrupt regular network service. This is why bandwidth measurements are performed rarely – and only before a service is in production – to test the path, or where there is a need to debug a network performance

[1] In recent years perfSONAR deployments on virtual machines or docker containers became common.

problem. On the other hand, gathering metrics such as packet loss, latency and latency variation continuously does not require high-end dedicated hardware and adds minimal overhead to the network traffic. Also, these three QoS metrics indirectly provide insights into any potential congestion in the network and their observed values are most often a direct consequence of congestion and buffering on some of the links on the path of a packet. Packet loss, latency and latency variation can reliably be monitored using standard OWAMP and TWAMP protocols, and small form factor devices.

Although it is possible to monitor all key performance metrics using small-footprint hardware or even virtual machines, this approach still assumes that additional equipment (a small probe or a server to host the software or a virtual machine) will have to be installed and maintained in network PoPs. This increases the complexity of the PoP configuration and the number of potential in-person interventions which can be a problem in case of remote network PoPs. For network operators, the easiest solution is to monitor the network and services using the tools available on network elements.

Until recently, equipment vendors used their proprietary non-interoperable solutions, e.g. Juniper RPM and Cisco IP SLA. However, in the last couple of years, new features have started to emerge on the most commonly used equipment, including support for standard interoperable protocols such as TWAMP and the possibility to install virtual services on network elements with either an operating system such as Ubuntu or a dedicated monitoring tool such as perfSONAR which use standard protocols. Using such features can enable detailed per-segment network service performance monitoring without the use of additional equipment – this is the zero-footprint monitoring approach.

The purpose of this document is to show capabilities and limits in using zero-footprint performance monitoring tools on various network elements, to present the results of the interoperability tests conducted between them and to provide guidelines for typical usage scenarios. This document aims to provide a valuable source of information and a guideline for NRENs who wish to set up or redesign their performance monitoring systems. It also includes examples of configurations for the tools presented; While these tools can be configured manually via a CLI, there are clear benefits to adopting automation, e.g. in enabling greater efficiency and in providing configuration integrity. A discussion of the progress made towards Automation and Orchestration of Services in the GÉANT Community can be found in Deliverable D6.2.

This document is structured as follows: Section 2 introduces basic information about the TWAMP protocol which is a de-facto standard for performance verification in IP networks. It describes the TWAMP capabilities of Juniper and Cisco devices, their limitations and configuration methods. The novel streaming telemetry method for sending measurement results towards result repositories where they are then available to visualisation tools is described as it applies to both vendors' equipment. Section 2 also describes how to use virtual services (e.g. Linux VM) on top of Cisco routers. Section 3 addresses possible data collection on results repositories using the novel streaming telemetry mechanism. Finally, Section 4 describes the use of TWAMP monitoring and interoperability in a multi-vendor use-case, set up in the GÉANT GTS environment.

2 TWAMP Protocol

The Two-Way Active Measurement Protocol (TWAMP) is an active measurement protocol defined in RFC5357. It is an extension of the older One-Way Active Measurement Protocol (OWAMP) specification which is capable of monitoring packet loss, latency and latency variation (jitter) between two hosts in each direction separately (one-way property). TWAMP adds two-way (round-trip time) measurements to the OWAMP monitoring portfolio. It establishes a control connection between the two devices which run a TWAMP client and a TWAMP server (TWAMP endpoints) and separate test sessions where the probe packets are exchanged. Since OWAMP and TWAMP measure one-way performance metrics, the endpoints must be time-synchronised with both protocols, which can be an additional challenge if particularly accurate measurements are desired. Recently, TWAMP became a common part of the feature sets of the largest network equipment vendors. This enables performance measurements between the devices of different vendors (interoperability), which was not previously possible.

Figure 2.1 below shows the key logical entities in TWAMP, including a Server which listens for measurement requests that come from the Clients (Session Senders). Probe packets are exchanged in both directions after the Control connection is established and monitoring configured.

```
+----------------+                +-------------------+
| Control-Client |<--TWAMP-Control-->|      Server       |
|                |                |                   |
| Session-Sender |<--TWAMP-Test----->| Session-Reflector |
+----------------+                +-------------------+
```

Figure 2.1: TWAMP architecture

The TWAMP protocol has recently been implemented by leading router vendors such as Juniper and Cisco. The Juniper implementation includes both a TWAMP server and a client, which allows monitoring of network latency in a per-segment way between a pair of Juniper routers, one of which runs a TWAMP client and the other a TWAMP server. The measurement data are calculated by a TWAMP client.

So far, Cisco have implemented only the TWAMP server in their routers so an external TWAMP client is needed to perform latency measurements. One possible solution for measuring delays on a per-segment basis between Cisco routers could be to use the twping client from the perfSONAR toolkit, which has TWAMP sessions with all Cisco TWAMP servers on routers along a path. Another possible solution might be to install a virtual machine on a Cisco router (currently only some models support it) and then a TWAMP client on the VM (e.g. twping from perfSONAR). Furthermore, Cisco has introduced Guestshell, a virtualised Linux-based environment that may also be used for running TWAMP clients.

2.1 Juniper TWAMP

This section briefly explains the way TWAMP can be configured and used on Juniper devices. Specifically, it provides information on how to configure TWAMP Juniper devices as TWAMP servers/clients and further explains how TWAMP measurements can be retrieved. The configuration may differ from one platform to another and depend on the software version. For the most accurate information about router configuration, the reader should refer to the configuration guides for their platform. More details about TWAMP on Juniper devices are available at the following link.

2.1.1 TWAMP Configuration

Figure 2.1 gives an example of the TWAMP server and client configuration on Juniper devices. Server configuration is simpler as it defines only the port on which the server listens to the requests and the set of IP addresses from which it expects the TWAMP requests (in this case port number 862).

```
server {
    authentication-mode none;
    port 862;
    client-list clietnts1 {
        address {
            10.0.0.0/8;
            172.16.0.0/16;
        }
    }
}
```

Figure 2.2: TWAMP Server configuration example

The TWAMP Client configuration is slightly more complex, as shown in Figure 2.3, as it requires the IP address of the Server it wants to talk to be defined, as well as the intervals between the measurements, number of measurements, number of probe packets and so on. The measurement results can be obtained by reading the output of the CLI commands and/or the appropriate SNMP variables, or through streaming telemetry as described in Section 2.1.2.2.

```
client {
    control-connection c23 {
        destination-interface si-0/0/0.20;
        history-size 500;
        routing-instance NetMon-ALL;
        target-address 10.4.3.1;
        test-count 4294967290;
        test-interval 1;
        traps {
            control-connection-closed;
        }
        test-session t23 {
            target-address 10.4.3.1;
            data-fill-with-zeros;
            probe-count 20;
            probe-interval 1;
        }
    }
}
```

Figure 2.3: TWAMP Client configuration example

2.1.2 Data Export

The results of TWAMP measurements can be transferred from the Juniper routers in two different ways to export data to an external collector:

- By polling appropriate Object Identifiers (OID)s from SNMP Management Information Base (MIB) objects (described in the next section) or
- By pushing data from the network device via the Juniper Streaming Telemetry Interface (described in Section 2.1.2.2).

2.1.2.1 *SNMP polling*

This section provides a brief explanation of the key data that can be found in the TWAMP MIB, which is a proprietary Juniper MIB. The terminology used in the Juniper TWAMP MIB is the following:

- '*Probe sample*' – a latency measurement of an individual probe packet, i.e. a packet generated by a TWAMP client.
- *'Test'*: a *set* of probe packets that belong to one instance of a *test-session.* Each test session starts with a new TCP connection setup between a client and a server.
- '*Set*' is used to refer to some probe packets/samples of the test or a sequence of tests. The MIB defines 4 types of sets in the variable JnxTwampClientCollectionType (=jnxTwampResSumCollection):
  - currentTest (1[2]) – the test currently being executed, which likely consists of less than probe-count probes, e.g. less than 55 in the given example
  - lastCompletedTest (2) – the most recently completed test; if there was no packet loss this consists of probe-count probes
  - movingAverage (3) – the 'n' most recent probes (n is configurable)
  - allTests (4) – all the probes (since the entry was last reset).

The TWAMP MIB consists of 8 tables:

- 3 tables of current results
- 3 tables of historical data
- 2 tables of the configuration parameters of a control-connection and test-session

Calculated Results Table 'jnxTwampClientResultsCalculatedTable'

This table stores the statistics (such as average, min, max, StdDev) calculated from the set of probe packets for all 4 types of sets defined by the jnxTwampResSumCollection index of the table entry. The most useful of these seems to be lastCompletedTest (2) set, as it represents statistics from the full set of probe packets of the latest completed test (hence out of all 55 packets of the example config).

[2] The type of this MIB object is integer. The numbers given in these lines are the values that can be obtained from the device, and each line contains the name and the description of the object.

The Calculated Results entry consists of the following statistics:

- jnxTwampResCalcSamples (2) – The number of samples used in these calculations
- jnxTwampResCalcMin (3) – The minimum of all the samples in the set (in microseconds as all the other times)
- jnxTwampResCalcMax (4) – The maximum of all the samples in the set
- jnxTwampResCalcAverage (5) – The average of all the samples in the set
- jnxTwampResCalcPkToPk (6) – The difference between the minimum and maximum of all the samples in the set
- jnxTwampResCalcStdDev (7) – The standard deviation of all the samples
- jnxTwampResCalcSum (8) – The sum of all the samples

There are 4 indexes that define what a table entry statistic represents:

- pingCtlOwnerIndex
- pingCtlTestName
- jnxTwampResSumCollection (=JnxTwampClientCollectionType)
- inxTwampResCalcSet (= JnxRpmMeasurementSet)

The first two indexes are text strings that define the test owner and test name (i.e. c23 is a test owner and t23 is a test name in the TWAMP client configuration example in Figure 2.3). The 3rd index jnxTwampResSumCollection has already been described above.

The 4th index defines the metrics the above statistics belong to, i.e.:

- roundTripTime (1) – the set of round-trip delays
- posRttJitter (2) – positive round-trip jitter measurements
- negRttJitter (3) – negative round-trip jitter measurements
- egress (4) – outgoing (source-to-destination) one-way delays
- posEgressJitter (5) – positive egress jitter measurements
- negEgressJitter (6) – negative egress jitter measurements
- ingress (7) – incoming (destination-to-source) one-way delays
- posIngressJitter (8) – positive ingress jitter measurements
- negIngressJitter (9) – negative ingress jitter measurements

In other words, for each metric a TWAMP agent calculates all 6 statistics, e.g. for roundTripTime it calculates Min, Max, Average, PkToPk, StdDev, and Sum.

The metrics set consists of two groups: delay metrics (roundTripTime, egress, ingress) and jitter metrics (posRttJitter, negRTTJitter, posEgressJitter, negEgressJitter, posIngressJitter, negIngressJitter). Delay metrics are measured according to the RFC 7679 definition, i.e. as time passed since the moment when the first bit of a packet was sent by Src and the moment when the last bit of the packet was received by Dst. Jitter metrics are measured according to the RFC 3393, i.e. as the difference between the current and the previous round-trip or one-way delays.

Positive and negative jitter statistics are calculated separately to avoid their mutual compensation. It seems both Jitter group of metrics and their statistics, and Min, Max, and StdDev statistics of the delay metrics are useful towards the evaluation of packet transmission timing The measurements carried out by the Incubator activity showed that the graphs of these two groups of statistics could be very close, with the difference between them no greater than 1-1.5 ms.

An example of the Calculated Table entry obtained by polling the TWAMP MIB is given in Figure 2.4:

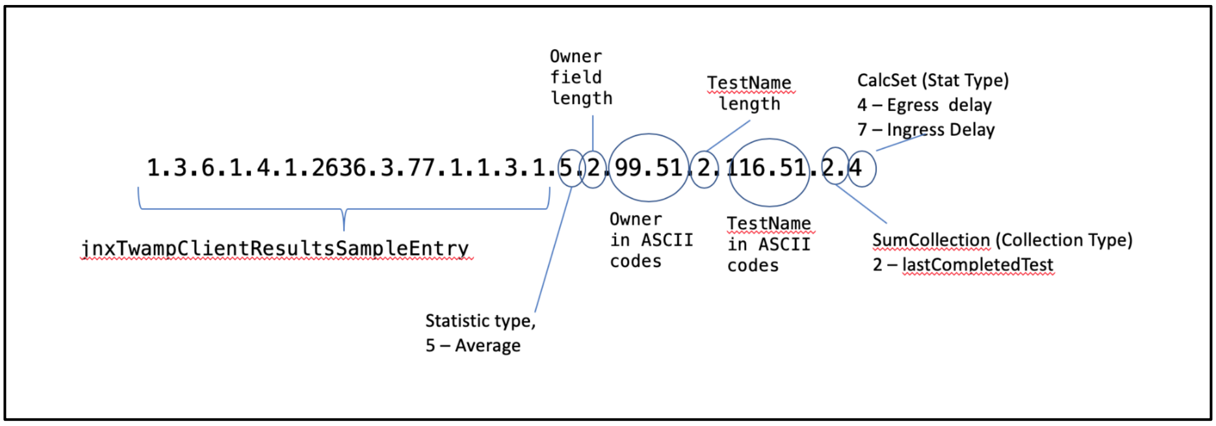


Figure 2.4: Calculated Table entry format

The Calculated Table does not have an object showing the Date/Time of the set measurement. This value can be obtained from the Summary Results Table.

The Summary Results Table ‘jnxRpmResultsSummaryTable’ stores such entries as the number of Sent, Received and Lost packets of the set, for which data are calculated and stored in the Calculated Results Table.

The table entry has four objects:

- jnxRpmResSumSent
- jnxRpmResSumReceived
- jnxRpmResSumPercentLost
- jnxRpmResSumDate.

The last of these represents the Date/Time of the set measurements.

The Summary Results table has three indexes, corresponding to the first three indexes of the Calculated Results Table.

2.1.2.2 *Streaming Telemetry*

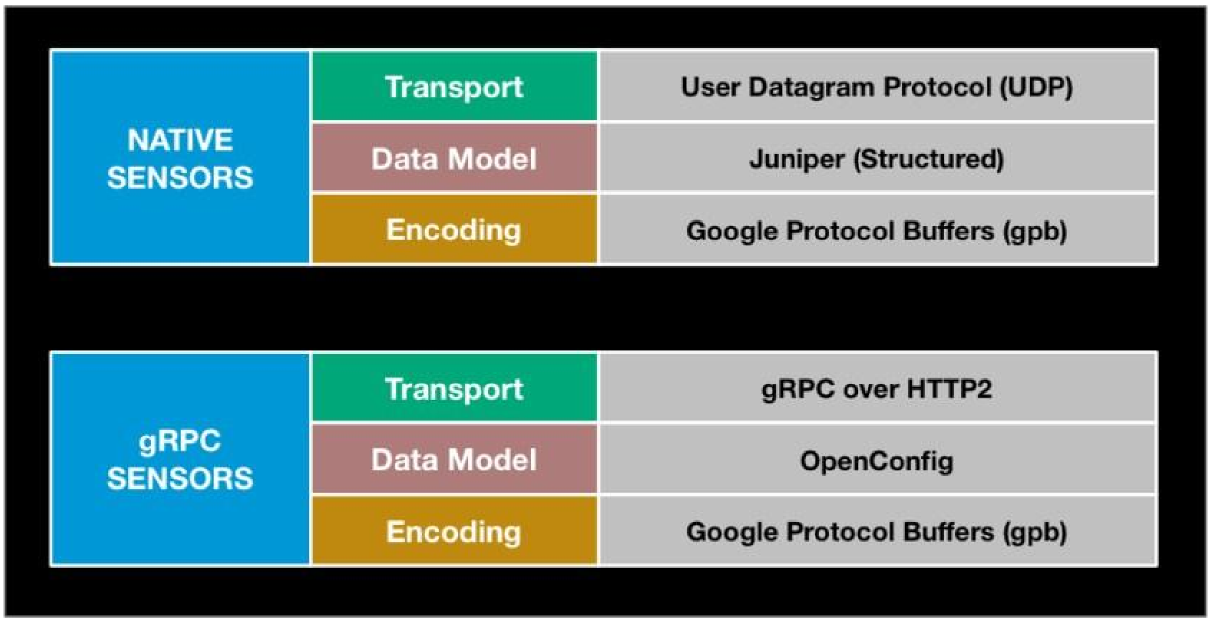
The Junos Telemetry Interface (JTI) supports two ways of providing streaming data from Juniper network devices:

1. **Native sensors:** these are line-card or PFE (Packet Forwarding Engine)-based sensors, which export the data via User Datagram Protocol (UDP). These sensors use Juniper's proprietary but open data model, using Google Protocol Buffers (GBP) to structure and serialise the

telemetry data for transmission. Each Native sensor has an associated Protocol Buffer file (or ".proto" file) which defines the content and structure of that sensor's telemetry payload.

2. **gRPC sensors:** these are Routing Engine (RE)-based sensors, which export the data using gRPC over HTTP2. These sensors use a data model defined by OpenConfig, and structure the telemetry data using a key/value pair format. Similar to Native sensors, gRPC sensors also employ Google Protocol Buffers.

A summary of the differences between these two sensor types is shown in Figure 2.5. When comparing these two telemetry formats, it can be seen that they both serialise the data in binary format using Google Protocol Buffers for transmission over the wire, but they differ in how the underlying telemetry data is encoded. For gRPC sensors, the underlying data is structured in a key/value pair format. This means that once the binary data off the wire is decoded (using Protobuf Compiler, or "protoc"), the underlying data is "self-describing" and the content can immediately be discerned as a listing of keys and associated values. Conversely, with Native sensors, once the binary data off the wire is decoded, the underlying data is still structured as Google Protocol Buffer messages and the associated .proto files must be used as a "secret decoder ring" in order to make sense of the underlying data. The more recent trend is towards the open gRPC format, so it is expected that implementations will migrate towards the open gRPC approach.



NATIVE SENSORS	Transport	User Datagram Protocol (UDP)
	Data Model	Juniper (Structured)
	Encoding	Google Protocol Buffers (gpb)
gRPC SENSORS	Transport	gRPC over HTTP2
	Data Model	OpenConfig
	Encoding	Google Protocol Buffers (gpb)

Figure 2.5: Native vs. gRPC telemetry

2.1.2.3 *Router Configuration*

This section describes the information on configuring gRPC Sensors on Juniper devices (in the case described in this document, vMXes with junOS version 19.3). The focus is given specifically to gRPC (gNMI) since Native Sensors did not provide TWAMP related metrics which are the target of this use case.

Before proceeding with router configuration details, the following prerequisites should be met:

- The Juniper Networks device that will be used should have Junos OS Release 16.1R3 version or later.
- If the Juniper device is running a version of Junos OS with an upgraded FreeBSD kernel, the Junos Network Agent software package should be installed (already included in 19.3 on vMX).
- OpenConfig for Junos module should also be installed (also included in 19.3 on vMX).

In order to configure gRPC services in Juniper devices, the following steps should be followed:

1. A self-signed certificate is required which can be created in a Linux-based machine using the following command:
   `openssl req -x509 -sha256 -nodes -newkey rsa:2048 -keyout cert.pem -out cert.pem`
2. The self-signed certificate needs to be transferred to the employed device e.g.: `file copy scp://172.16.0.252//home/gts/streaming_telemetry/vMX2/cert.pem /var/tmp/` where 172.16.0.252 is the IP address of the system with the certificate
3. The certificate needs to be loaded to the Juniper device:
   `set security certificates local jti-cert load-key-file cert.pem`
4. gRPC can also be configured without SSL and as a consequence without using a certificate, however in tests on vMX 19.3 the router did not allow proceeding with the gRPC configuration without using SSL
5. After loading the certificate the following command was used to define that SSL will be used, and the port that listens for subscribers is port `32767`:
   `set system services extension-service request-response grpc ssl port 32767`
6. Optionally the following command can be used to allow only a subnet to retrieve data from the device via gRPC:
   `set system services extension-service notification allow-clients address 172.16.0.0/24`

A configuration file based on the aforementioned commands is given in Figure 2.6.

```
system {
    services {
        ssh;
        telnet;
        extension-service {
            request-response {
                grpc {
                    ssl {
                        port 32767;
                        local-certificate vMX1;
                    }
                }
            }
        }
    }
```

Figure 2.6: gRPC Sensors configuration on Juniper devices

2.2 Cisco TWAMP and IP SLA

The TWAMP server/responder feature is available for IOS, IOS XE and IOS XR versions that were released in the last couple of years. The instructions given in this section are based on Cisco IOS XE 16.12 on CSR1000v routers. The configuration may differ from one platform to another and depend on the software version. For the most accurate information about the router configuration, the reader should refer to the configuration guides for their platform

On the CSR1000v platform, Cisco supports the TWAMP protocol only in the receiver (server) mode. This means that TWAMP measurements cannot be initiated from Cisco routers, and TWAMP measurement results are not stored on them. Therefore, Cisco refers to its implementation as "TWAMP responder". There are two ways to use TWAMP on current Cisco implementations:

1. Initiate measurements from other devices (e.g. Linux hosts or Juniper routers), and get the measurement results from the initiators (described in the sections 2.2.1 and 2.2.2) or,
2. Install a virtual service on a Cisco router which is capable of being the TWAMP initiator. An example of an Ubuntu machine installed on a Cisco router as a virtual service which runs the TWAMP code is given in Section 2.2.3.

On the other hand, Cisco supports its proprietary IP SLA feature set which allows various performance measurements in both directions. The next sections show how to install TWAMP on a Cisco router and how to set up some basic UDP jitter monitoring using the IP SLA feature.

2.2.1 Performance Monitoring Configuration

2.2.1.1 *TWAMP Responder Configuration*

TWAMP responder is configured as in the example given in Figure 2.7. The key configuration parameter is the port number which in this example is 9000. To get TWAMP results between a Linux host and a Cisco router, the following command can be used on the Linux machine: `twping router_ip_address:9000`.

```
ip sla responder
ip sla responder twamp
 timeout 300
ip sla server twamp
 port 9000
 timer inactivity 300
```

Figure 2.7: Cisco TWAMP responder configuration

2.2.2 Data Export

A classic, well-known method to get performance measurement results is to retrieve them from the Command Line Interface (CLI), or by polling the appropriate SNMP Object Identifier (OID). However, in this section, the emerging streaming telemetry approach is presented.

2.2.2.1 *Cisco Model-Driven Telemetry*

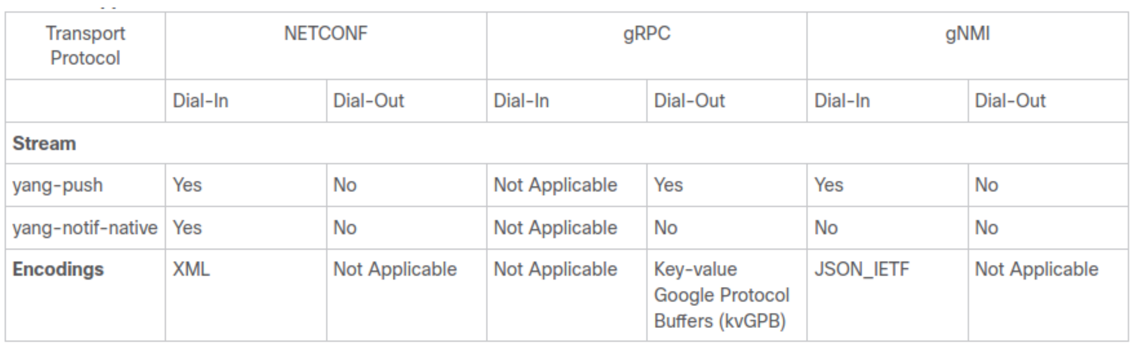
Cisco Model-Driven Telemetry (MDT) is a mechanism to stream YANG-modelled data from a router to a data collector. There can be several entities in the MDT:

- Publisher – a network element that streams the data
- Receiver/collector – a device that receives data
- Controller – a device that creates subscriptions for some data streams but does not receive the data (tells the Publisher to send the data to the Receiver)
- Subscriber – a device that creates subscriptions for some data streams and is a data receiver.

Cisco supports two types of subscriptions:

- Dynamic – subscriptions configured by a Subscriber – "**dial-in**" subscriptions. Dial-in subscriptions are not stored in the Publisher configuration file and must be restarted by the Subscriber upon the Publisher reset.
- Configured – which data to stream, and where to stream them, is configured on the Publisher – "**dial-out**" subscriptions. Dial-out subscriptions are configured in the Publisher CLI and can be stored in the startup configuration.

Cisco supports NETCONF, RESTCONF and gNMI northbound interfaces to stream data. Table 2.1 (source: MDT) summarises which transport protocols are being used for different types of subscriptions.



Transport Protocol	NETCONF		gRPC		gNMI	
	Dial-In	Dial-Out	Dial-In	Dial-Out	Dial-In	Dial-Out
Stream						
yang-push	Yes	No	Not Applicable	Yes	Yes	No
yang-notif-native	Yes	No	Not Applicable	No	No	No
Encodings	XML	Not Applicable	Not Applicable	Key-value Google Protocol Buffers (kvGPB)	JSON_IETF	Not Applicable

Table 2.1: Transport protocols in use for different types of subscriptions

2.2.2.2 *Enabling NETCONF*

The key prerequisite for MDT to work is that NETCONF-YANG is enabled. NETCONF-YANG is enabled using the following command:

```
   XE1(config)#netconf-yang
```

NETCONF requires authentication of the user who accesses the router. The user must have privilege level 15. The router can either use existing users that are defined on a router or a new credential must be configured for NETCONF. In addition, it is possible to configure a "candidate datastore" feature. The candidate datastore provides a temporary workspace in which a copy of the device's running configuration is stored. This feature resolves problems (configuration overwrites) in those situations where there are multiple users working simultaneously on a router configuration. NETCONF uses default port 830, although this parameter is configurable.

NETCONF status can be verified using the command given in Figure 2.8.

```
XE1#show platform software yang-management process
confd                : Running
nesd                 : Running
syncfd               : Running
ncsshd               : Running
dmiauthd             : Running
nginx                : Running
ndbmand              : Running
pubd                 : Running
```

Figure 2.8: NETCONF status verification

ncsshd is a NETCONF SSH daemon, which is running in this example. An additional test could be gathering NETCONF device capabilities (supported YANG models) using the command in Figure 2.9. As the Cisco CSR1000v platform with IOS XE 16.12 has more than 450 capabilities, a shortened example is provided.

```
H5:~$ ssh -s gts@172.16.0.82 -p 830 netconf
gts@172.16.0.82's password:
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
<capability>urn:ietf:params:netconf:base:1.1</capability>
...
<capability>http://cisco.com/ns/yang/Cisco-IOS-XE-ip-sla-oper?module=Cisco-IOS-
XE-ip-sla-oper&revision=2019-05-01</capability>
...
<capability>urn:ietf:params:xml:ns:netconf:base:1.0?module=ietf-
netconf&revision=2011-06-01</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-netconf-with-defaults?module=ietf-
netconf-with-defaults&revision=2011-06-01</capability>
<capability>
        urn:ietf:params:netconf:capability:notification:1.1
      </capability>
</capabilities>
<session-id>442</session-id></hello>]]>]]>
```

Figure 2.9: Gathering available NETCONF capabilities from a Cisco device

2.2.2.3 *Configuring Dial-Out Data Streaming*

The last step is to configure the push of the data stream from the router towards the data collector. This configuration is shown in Figure 2.10, and the key parameters are:

- **xpath** of the variables that are streamed
- The address of the router from which the data is streamed (**source-address**).
- The rate of variable stream (**update policy**) in ms. The update policy can also be configured to be set to "update-policy on-change", in which case the data is streamed only if the parameters change
- The IP address, port number and protocol of the data store (**receiver**).

```
  telemetry ietf subscription 98
   encoding encode-kvgpb
   filter xpath /ip-sla-ios-xe-oper:ip-sla-stats
   source-address 172.16.0.82
   stream yang-push
   update-policy periodic 5000
   receiver ip address 172.16.0.252 57000 protocol grpc-tcp
```

Figure 2.10: Telemetry subscription configuration

The given router configuration process is relatively straightforward and well described in Cisco's configuration guides. What is not so obvious at first is how to locate the correct xpath for the variables to stream to the data collector. One method for doing this is described in Appendix A.

2.2.3 Cisco Virtual Services

Cisco IOS XE supports hosting network services and applications within network devices such as ISR4000, ASR1000 or CSR1000 routers. Third-party Kernel-based Virtual Machine (KVM) applications can easily be hosted directly on such networking devices. The application runs in the virtual services container of the operating system of a device. It is delivered as an Open Virtual Application (OVA), which is an especially prepared tar file with an .ova extension. The OVA package is installed and enabled on a device through the device CLI. This architecture is shown in Figure 2.11. The control plane is entirely Linux running on an x86 CPU from Intel. Since the control plane of most routers is not completely used, this opens up the possibility for hosting KVMs directly on spare CPU cycles.

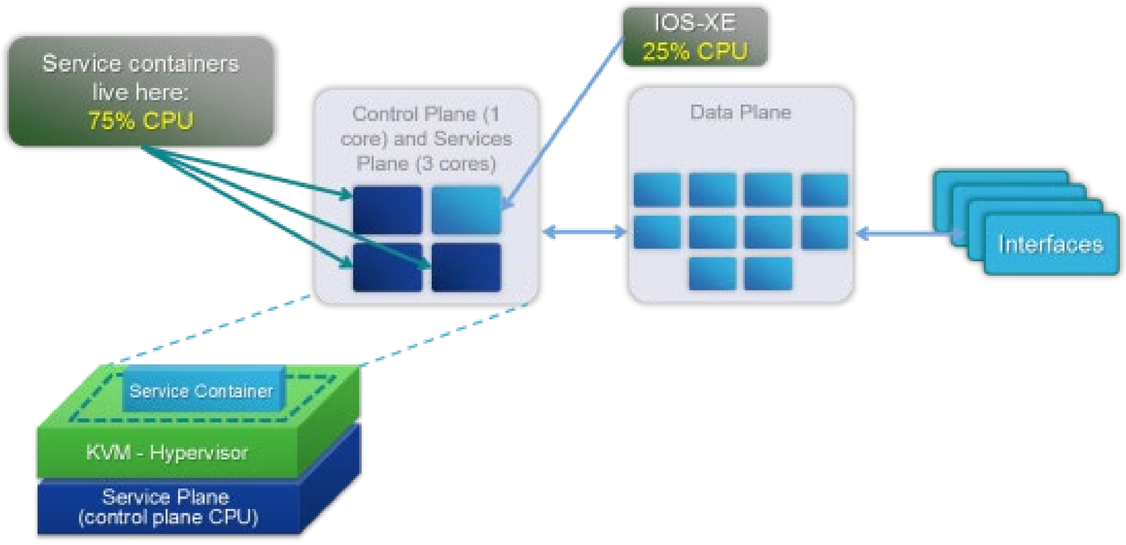

Service containers live here: 75% CPU
IOS-XE 25% CPU
Control Plane (1 core) and Services Plane (3 cores)
Data Plane
Interfaces
Service Container
KVM - Hypervisor
Service Plane (control plane CPU)

Figure 2.11: Cisco virtual service architecture[3]

Since the virtual services are not hosted on the data plane, there is no impact on performance in terms of router packet forwarding. Control plane functions that handle routing and other device-related control functions run at a higher priority than hosted virtual machines. However, any time the router does not need all the resources reserved for control plane functions, the excess CPU and memory resources can be used for virtual services.

[3] Image taken from: https://blogs.cisco.com/networking/kvm-app-hosting-on-a-cisco-router

With this feature it is possible to install a Linux virtual machine on top of a Cisco router. This virtual machine can be equipped with the appropriate monitoring software (e.g. perfSONAR Tools bundle) and used to initiate twamp measurements.

The key steps in the virtual service installation and configuration are:

- Preparing the OVA image
- Virtual service installation and configuration
- Monitoring agent installation and configuration

Information on preparing the appropriate virtual service image is given in Appendix B, while the other two activities are described below.

This section presents an example of the installation, configuration and usage of an Ubuntu 18.04.5 LTS virtual machine installed as a virtual service on top of a Cisco router. Only the most relevant part of the configuration is described here; the routing protocols, SNMP and other feature configurations are omitted. The logical connection of the Ubuntu VM is given in Figure 2.12. On the CSR1000v, the VM is connected to the VirtualPortGroup0. It is possible to have various configurations with a fixed IP address or the IP address of another interface (e.g. an unnumbered interface). In the example given, a fixed IP address – 10.2.200.1/24 – is used on the VirtualPortGroup0 interface and the address of the VM connected to that interface is 10.2.200.2/24. Since the testbed as well as the VM are in the IPv4 private address range, NAT was configured between the VPG0 and Gi1 interfaces, and not towards Gi2.

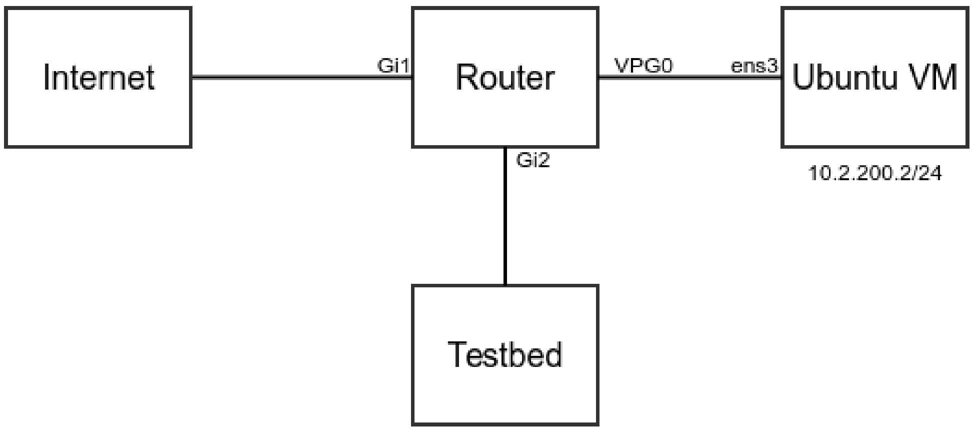


Figure 2.12: Ubuntu virtual service on Cisco router – configuration

The part of the configuration below shows its key elements: NAT between the VPG0 and Gi1 interfaces and the commands to run the virtual service. The name of the virtual service in this case is UBUNTU and the system will recognise it under that name.

Service VM configuration

```
interface VirtualPortGroup0
 ip address 10.2.200.1 255.255.255.0
 ip nat inside
 ip router isis
!
interface GigabitEthernet1
 description MANAGEMENT
 ip dhcp client client-id ascii 9LDTNFMDSOS
 ip address dhcp
 ip nat outside
!
virtual-service
 signing level unsigned
!
virtual-service UBUNTU
 vnic gateway VirtualPortGroup0
!
ip nat inside source route-map NAT_TO_Internet interface
GigabitEthernet1 overload
!
ip access-list standard 1
 10 permit 10.2.200.0 0.0.0.255
!
route-map NAT_TO_Internet permit 10
 match ip address 1
 match interface GigabitEthernet1
!
```

After the configuration is entered, the service VM can be installed with the following command, assuming the name of the OVA file is bionic-server.ova:

```
virtual-service install name UBUNTU package bootflash:bionic-server.ova
```

It is then necessary to wait for the virtual service to be installed. The status should be "Installed":

```
show virtual-service list
Virtual Service List:

Name                     Status             Package Name
------------------------------------------------------------------------------
UBUNTU                   Installed          bionic-server.ova
```

Then activate the virtual machine by configuring the following:

```
XE1(config)#virtual-service UBUNTU
XE1(config-virt-serv)#activate
```

The status should be:

```
show virtual-service list
Virtual Service List:

Name                     Status             Package Name
------------------------------------------------------------------------------
UBUNTU                   Activated          bionic-server.ova
```

Finally, connect to the machine using the following command:

```
virtual-service connect name UBUNTU console
```

The CLI interface of the Ubuntu machine should then appear. The exit sequence is ^C ^C ^C. The next step is to configure the IP address on the ens3 port, and then the machine can be accessed from the other devices via SSH.

After the Ubuntu virtual machine is enabled, it is necessary to configure the interface of the virtual machine (in this case ens3), and once this is connected to the Internet, it is possible to update and upgrade the software or to install any software tool for that platform. As described in Section 4, the network monitoring software suite that was used to run the TWAMP performance tests was the perfSONAR-tools bundle. Data export from the virtual service can be organised in the same way as for any other Linux machine.

3 Streaming Telemetry Data Collection

Collecting different types of data from various vendors' equipment in a unified manner poses significant challenges. To that end, Telegraf (v1.14.2), a software package for collecting and reporting metrics for all kinds of data with a low memory footprint, was used in the setup described in Section 4 to extract the generated monitoring data from the network devices and store them (backend) in InfluxDB (v1.7.10).

The subsections below present how Telegraf can be configured to subscribe to Juniper devices (2.1.2) and collect data from Cisco devices (2.2.2), leveraging the streaming telemetry capabilities of the network devices. gRPC sensors are focused on since gRPC is an open-source framework already embraced by a wide range of vendors, open-source projects and operators for network management solutions (and is not a vendor-specific format). Specifically for this use case, gRPC sensors can be configured to provide TWAMP related data in comparison to Juniper's native sensors that do not provide such functionality.

3.1 Juniper

gRPC Sensors

Telegraf (v1.14.2) supports the collection and storage (via InfluxDB) of Juniper OpenConfig Telemetry data. An indicative Telegraf configuration file for Juniper OpenConfig Telemetry data[4] is provided in Figure 3.1. The main parameters that can be defined are:

- The network devices that the collector subscribes to (line 4)
- The appropriate credentials for the subscription (line 9-10)
- The default frequency to ask for data (this can be also configured per monitored metric e.g. line 24)
- The sensors (monitoring data) that the collector subscribes to with optional identifiers (see line 25, ***twampmeasurements*** */junos/twamp/client/probe-test-results/*)

[4] https://github.com/Juniper/telegraf-jti-plugins/tree/master/plugins/inputs/jti_openconfig_telemetry

```
# # Read JTI OpenConfig Telemetry from listed sensors
 [[inputs.jti_openconfig_telemetry]]
#   ## List of device addresses to collect telemetry from
   servers = ["vMX1:32767","vMX2:32767","vMX3:32767","vMX4:32767"]
#
#   ## Authentication details. Username and password are must if device expects
#   ## authentication. Client ID must be unique when connecting from multiple
instances
#   ## of telegraf to the same device
   username = "gts"
   password = "Netmon"
   client_id = "telegraf"
#
#   ## Frequency to get data
   sample_frequency = "10000ms"
#
#   ## Sensors to subscribe for
#   ## A identifier for each sensor can be provided in path by separating with space
#   ## Else sensor path will be used as identifier
#   ## When identifier is used, we can provide a list of space separated sensors.
#   ## A single subscription will be created with all these sensors and data will
#   ## be saved to measurement with this identifier name
  sensors = [
   "/interfaces/",
   "collection /components/ /lldp",
   "twampmeasurements /junos/twamp/client/probe-test-results/"
  ]
#
#   ## We allow specifying sensor group level reporting rate. To do this, specify the
#   ## reporting rate in Duration at the beginning of sensor paths / collection
#   ## name. For entries without reporting rate, we use configured sample frequency
#   sensors = [
#    "1000ms customReporting /interfaces /lldp",
#    "2000ms collection /components",
#    "/interfaces",
#   ]
#
#   ## Optional TLS Config
    enable_tls = true
    #tls_ca = "/home/gts/streaming_telemetry/ca.pem"
#   # tls_cert = "/etc/telegraf/cert.pem"
#   # tls_key = "/etc/telegraf/key.pem"
#   ## Use TLS but skip chain & host verification
    insecure_skip_verify = true
#
#   ## Delay between retry attempts of failed RPC calls or streams. Defaults to
1000ms.
#   ## Failed streams/calls will not be retried if 0 is provided
   retry_delay = "1000ms"
#
#   ## To treat all string values as tags, set this to true
   str_as_tags = true
```

Figure 3.1: Telegraf Configuration

Takeaways for collecting streaming Juniper Openconfig telemetry data via Telegraf include:

- The collector can be configured to subscribe to a large list of data sources[5]
- Different reporting rates (depicting how frequently the data are pushed to the subscriber) can be configured per monitoring metric.

[5] https://www.juniper.net/documentation/en_US/junos/topics/reference/general/junos-telemetry-interface-grpc-sensors.html

- Custom names can be defined for the collected data

Especially for TWAMP related metrics, the collector can subscribe to the following paths:

Path	Available Data
/junos/twamp/client/control-connection/	**Data for the control connection of TWAMP measurements from the client side**
/junos/twamp/server/control-connection/	**Data for the control connection of TWAMP measurements from the server side**
/junos/twamp/client/probe-test-results/	**Data related to the TWAMP measurements e.g. RTT, Jitter**

Table 3.1: TWAMP metrics subscription paths

3.2 Cisco

As mentioned, Telegraf provides a unified interface to extract and store various data, including telemetry data, from Cisco devices. An indicative Telegraf configuration file for Cisco Model-Driven Telemetry (MDT) is provided in Figure 3.2. More information about MDT is available at the following link.

```
# # Cisco model-driven telemetry (MDT) input plugin for IOS XR, IOS XE and NX-OS
platforms
 [[inputs.cisco_telemetry_mdt]]
#   ## Telemetry transport can be "tcp" or "grpc".  TLS is only supported when
#   ## using the grpc transport.
  transport = "grpc"
#
#   ## Address and port to host telemetry listener
  service_address = ":57000"
#
#   ## Enable TLS; grpc transport only.
#   # tls_cert = "/etc/telegraf/cert.pem"
#   # tls_key = "/etc/telegraf/key.pem"
#
#   ## Enable TLS client authentication and define allowed CA certificates; grpc
#   ##  transport only.
#   # tls_allowed_cacerts = ["/etc/telegraf/clientca.pem"]
#
#   ## Define (for certain nested telemetry measurements with embedded tags) which
fields are tags
#   # embedded_tags = ["Cisco-IOS-XR-qos-ma-oper:qos/interface-
table/interface/input/service-policy-names/service-policy-instance/statistics/class-
stats/class-name"]
#
#   ## Define aliases to map telemetry encoding paths to simple measurement names
  [inputs.cisco_telemetry_mdt.aliases]
    ifstats = "ietf-interfaces:interfaces-state/interface/statistics"
```

Figure 3.2: Telegraf Configuration

4 Interoperability and Experimental Results

In this section, an example is provided that uses all the technologies and techniques mentioned in this document. The experiment described here had the following goals:

1. To test TWAMP interoperability between different vendors' devices.
2. To test per-link network service performance and compare it to similar measurement approaches, as employed by dedicated monitoring boxes.
3. To test the use of streaming telemetry for the collection of monitoring data.

4.1 Network and Infrastructure

The network used in the experiment was built on top of the GÉANT GTS infrastructure. The testbed had 13 Ubuntu virtual machines:

- Hosts H1-H10 and Monitor running Ubuntu16.04.6 LTS
- A UoC machine which is an Ubuntu 18.04.5 LTS virtual service on Cisco CSR1000v
- Hosts c1 and c2 running Ubuntu 18.04.5 LTS
- vMX1-vMX6 – 6 Juniper virtual MX (vMX) routers running junOS version 19.3, 2
- VSI1 and VSI2 – virtual openvswitch switches
- XE1 and XE2 – 2 Cisco virtual routers (CSR1000v) running Cisco IOS XE Software Version 16.12.03,

It also included several redundant links between the devices as depicted in Figure 4.1.

Resources for the testbed are located in 4 GÉANT PoPs across Europe which allows for a realistic evaluation of the obtained latency and jitter results, without the need to simulate network delays. Intradomain routing between the vMX and Cisco routers was established using the Intermediate System to Intermediate System (ISIS) protocol.

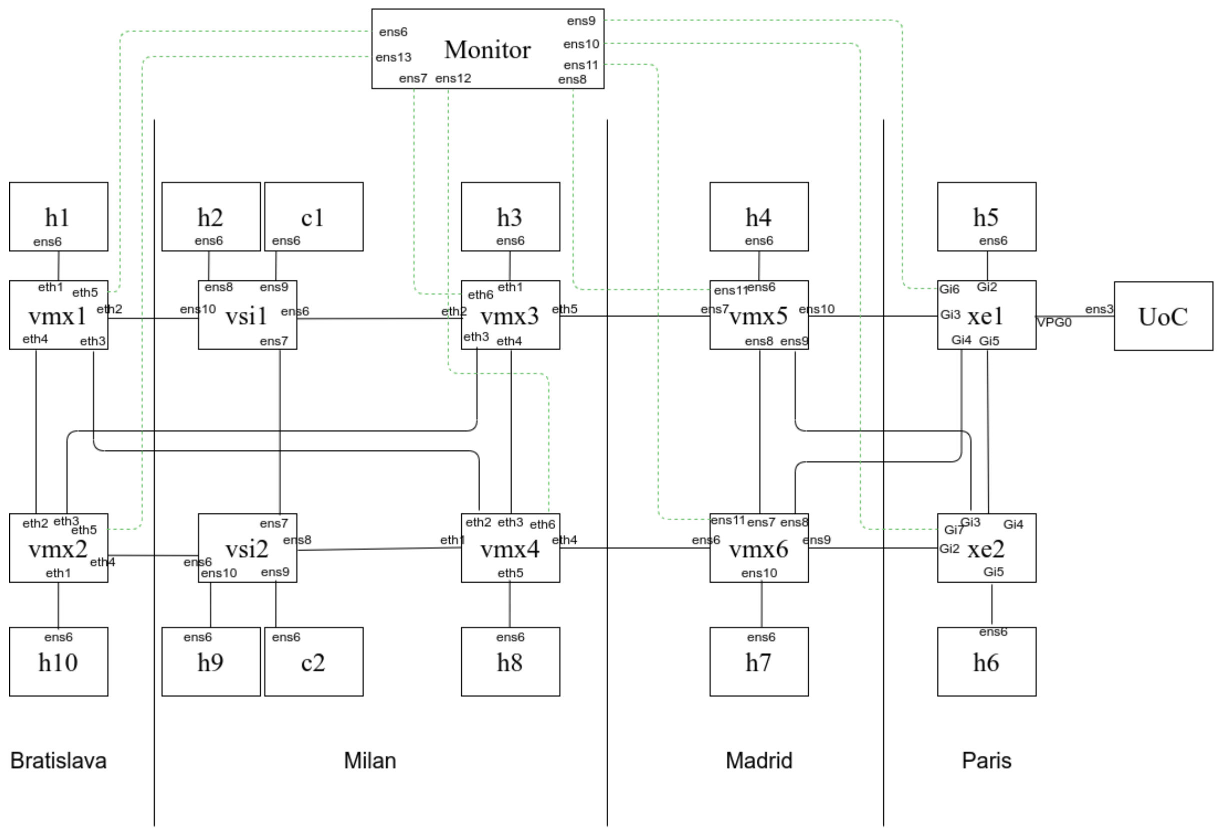


Figure 4.1: Interoperability testbed on GTS

All hosts H1-H10 had the perfSONAR-tools bundle installed including twping. The monitor virtual machine also included the following software components:

- Telegraf v1.14.2
- Grafana v6.7.2
- InfluxDB v1.7.10

4.2 Experimental Evaluation

The path that was tested and used for the experiment was between the H1 and H5 virtual machines and went through the Juniper vMX and Cisco XE devices, while at the endpoints H1 and H5 are Ubuntu virtual machines. In order to test the per-segment performance parameters, a set of measurements between the first hop (H1) and subsequent hops (vMX1, vMX3, XE1, H5) were performed. Since latency is an additive parameter, in this way it is possible to notice if there is an increase in the latency on some of the segments (e.g., by subtracting the one-way latency between the H1-vMX3 and H1-vMX1 pairs it is possible to infer the latency on the vMX1-vMX3 link) which is often an indication of congestion.

Host-to-host sessions rely on twping, part of the perfSONAR toolbox, while router-to-router sessions are based on the TWAMP implementation on the Juniper and Cisco network devices. Host-to-router tests were performed between the TWAMP software on Ubuntu (perfSONAR tools) and the TWAMP implementations on the Juniper and Cisco routers. The team also tested the use of the perfSONAR tools on a virtual machine installed as a virtual service on top of a Cisco router.

The results of the host-to-host and host-to-router TWAMP measurement were parsed and stored on the TWAMP Client, e.g. for H1- vMX1 and H1 - H3 the TWAMP Client is H1. These were then available after the TWAMP test was finished. For router-to-router sessions, the results were continuously retrieved from the TWAMP Client Router (in these experiments, every 1s), thus providing a high granularity of the monitored metrics.

The experiments performed focused on evaluating the monitoring services supported within different network devices on identifying:

1. Link quality deterioration, and
2. Rerouting scenarios.

In order to compare the results of the monitoring features of network elements with the results obtained from the measurements between the Linux hosts, the following TWAMP sessions were initiated:

- vMX1 - vMX3 (Juniper TWAMP -> Juniper TWAMP)[6]
- vMX2 - vMX1 (Juniper TWAMP -> Juniper TWAMP)
- vMX3 - vMX4 (Juniper TWAMP -> Juniper TWAMP)
- vMX4 - vMX2 (Juniper TWAMP -> Juniper TWAMP)
- vMX3 - XE1 (Juniper TWAMP -> Cisco TWAMP)
- H1 - H3 (perfSONAR tools TWAMP -> perfSONAR tools TWAMP)
- H1 - H5 (perfSONAR tools TWAMP -> perfSONAR tools TWAMP)
- H1 - vMX1 (perfSONAR tools TWAMP -> Juniper TWAMP)
- H1 - vMX3 (perfSONAR tools TWAMP -> Juniper TWAMP)
- H1 - XE1 (perfSONAR tools TWAMP -> Cisco TWAMP)

In addition to those above, the team also successfully tested the following TWAMP sessions:

- perfSONAR tools TWAMP on Cisco VM -> Cisco TWAMP
- perfSONAR tools TWAMP on Cisco VM -> perfSONAR tools TWAMP
- perfSONAR tools TWAMP on Cisco VM -> Juniper TWAMP

The results of all these measurements initiated between the various combinations of TWAMP clients and servers did not show any significant difference in the latency or jitter measurements between the devices in the same nodes, regardless of the TWAMP client or server choice. This indicates that all the TWAMP components tested are of similar reliability and accuracy.

[6] The notation used to describe TWAMP sessions is (TWAMP client -> TWAMP server)

In order to model a real-life scenario in the European NREN networks, arbitrary delay and jitter were introduced at these specific time windows:

15:00: Arbitrary delays are added using the tc tool on selected interfaces of the VS1 and VS2 devices:

- VS1 ens10: 20ms, ens6: 40ms
- VS2 ens6: 40ms, ens10: 60ms

15:10: Interface ens10 that connects vMX1 to VS1 is brought down and the traffic is redirected via vMX1-vMX2-VS2-vMX4-vMX3.

15:20: The link (ens10 up) is restored and network traffic is rerouted via the path: vMX1-VS1-vMX3.

15:30: Delays are returned to their original values.

15:40: The Jitter is added to the vMX1 interface (vMX1->vMX3).

15:50: The Jitter is removed from the vMX1 interface.

Delay Measurements

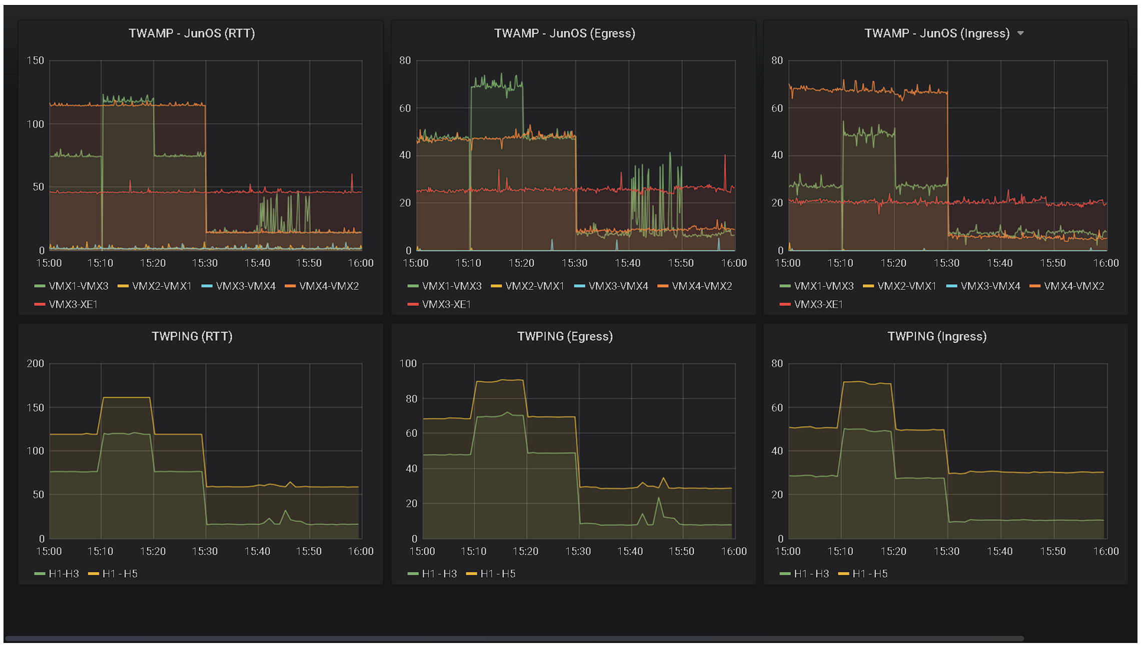


Figure 4.2: Delay measurements

As illustrated in Figure 4.2, both TWAMP session measurements from host-to-host and router-to-router provided the current state of the delay on each monitored path. At 15:10, a sharp increase in the delay can be observed, that is identified by both approaches. Considering only the host-to-host measurements, a sharp increase in the delay is observed between H1 and H3, which can be attributed to either congestion issues or a rerouting case. Observing the router-to-router measurements and specifically vMX1-vMX3, a sharp decrease in the delay can be seen that may infer that the resulting increase is related to a problematic link that went down and a different path that was chosen. At 15:20 the problematic link is restored and both approaches identify the decrease in the delay between vMX1 and vMX3; this is also the case at 15:30. For the anomalies (sharp increases/decreases) in delay and jitter, both approaches can pinpoint the direction (egress/ingress) in which the problem/issue occurred.

Jitter Measurements

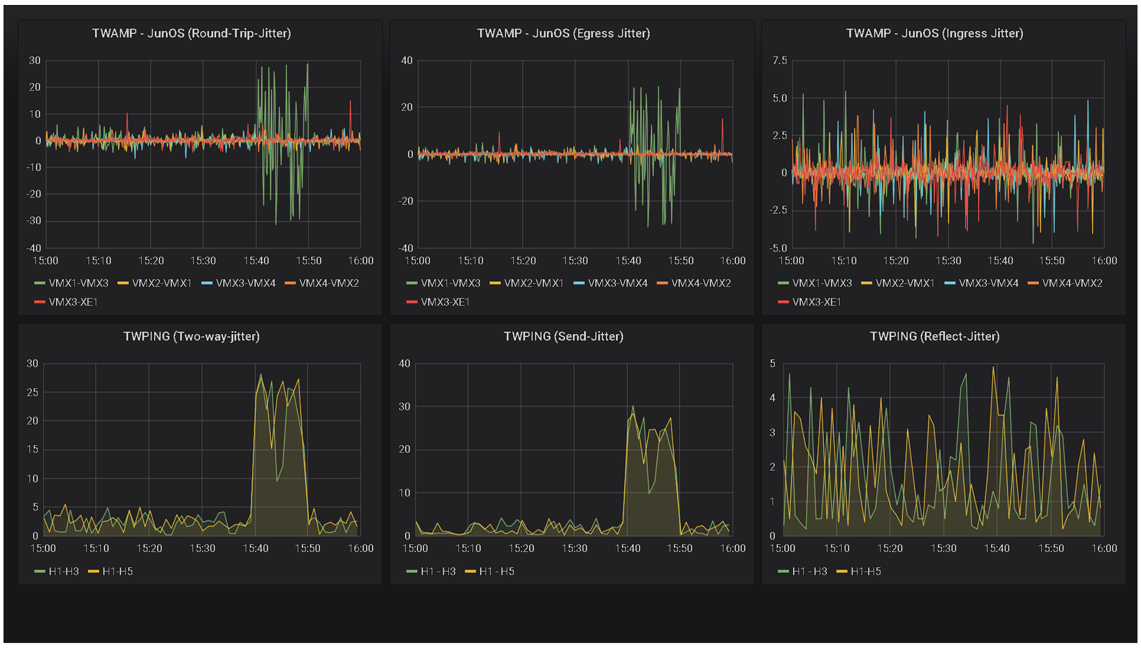


Figure 4.3: Jitter measurements

At 15:40 jitter was introduced in the vMX1 interface on the link between vMX1 and vMX3 (Figure 4.3). Similarly to what occurred with the delay results, it can be seen that the increased jitter is identified by both approaches almost immediately. Notably, both approaches can show in which direction the problem occurs. However, the router-to-router approach provides almost real-time measurements (the current jitter value is collected each second), while in the host-to-host case, it is necessary to wait for all the results to be retrieved by the TWAMP client. This means that for the whole duration of a TWAMP test in the host-to-host case there is no information related to the network state, so for example microbursts might be missed.

Host-to-Router Measurements

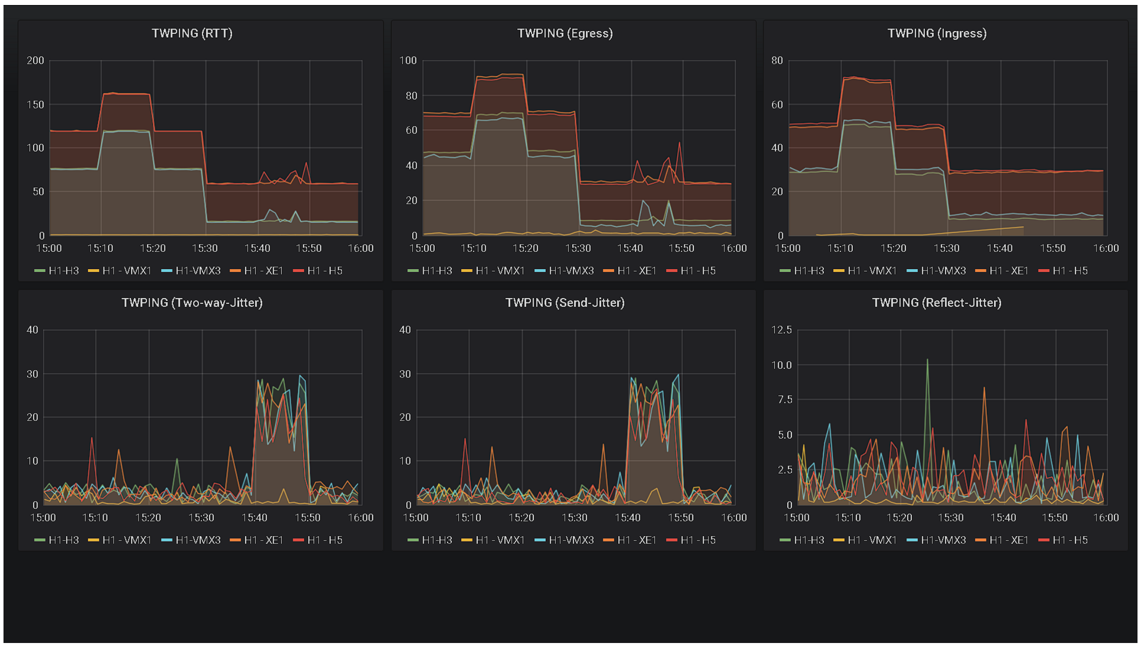


Figure 4.4: Host-to-router measurements

Finally, the host-to-host and router-to-router approaches can be mixed using a host-to-router measurement approach. Hosts initiate TWAMP sessions with the router TWAMP servers. As observed in Figure 4.4, this approach can also identify all the aforementioned anomalies that were introduced for both delay and jitter.

Another set of experiments between H1 and XE1 was conducted in order to verify whether there was any difference in the results obtained when the Cisco TWAMP server is used on XE1 compared to the case where a virtual service with a perfSONAR-tools implementation of TWAMP is used on top of XE1. No significant differences were observed indicating that both approaches give results of similar accuracy. What should be taken into account is the fact that the XE1 router in both cases was without any significant network traffic and CPU load.

5 Conclusions

This document gives an overview of the methods and technologies that could enable network operators to build a network of active monitoring probes without adding any additional hardware in their PoPs, an approach known as zero-footprint monitoring.

The reliability and interoperability of the different implementations of the RFC-based TWAMP monitoring code have now reached levels that ensure accurate network service performance evaluation even in multi-vendor environments. This means there is no longer a need to rely on the proprietary performance evaluation systems provided by some vendors. Specifically, the key conclusions that can be drawn from the tests presented here are:

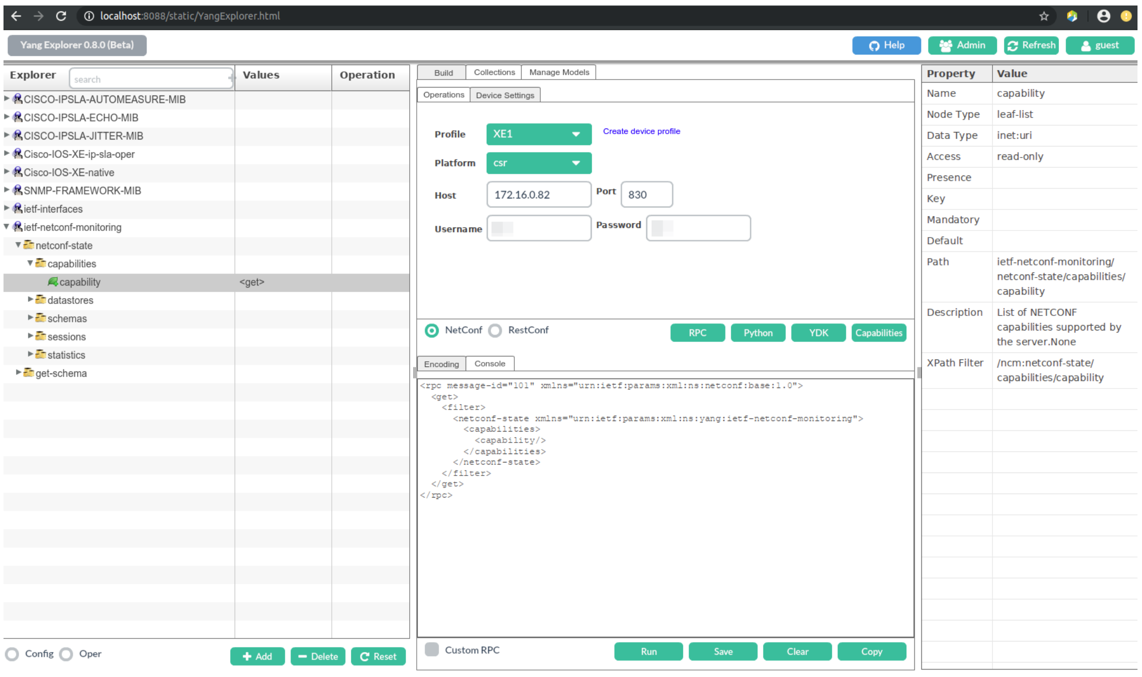
- Router-to-router TWAMP measurement empowered by streaming telemetry provides fine-grained real-time delay and jitter data.
- Host-to-host, host-to-router and router-to-router TWAMP sessions are capable of identifying delay and jitter fluctuations, also pinpointing problematic directions.
- TWAMP is interoperable between Cisco, Juniper, Ubuntu hosts (with the TWAMP implementation found in the perfSONAR suite) and an Ubuntu virtual service hosted on a capable Cisco router (with the same TWAMP implementation).
- An insight into per-segment latency degradation can be obtained by using the approach described in Section 4.2 where the performance parameters are measured between the first hop on the path and all successive network hops.
- Monitoring software installed on an Ubuntu virtual service hosted on a Cisco router provides reliable monitoring results.

In addition, streaming telemetry provides an efficient and now mature method for gathering measurement results from devices without the need to poll devices periodically.

Appendix A YANG explorer – Looking for the Xpath

YANG Explorer is a tool developed by Cisco which at the time of writing is still in a beta version, and which may never be made fully production-ready, as the latest changes in the code happened in early 2019. However, this is still a useful tool to explore the capabilities of a device and to get the xpaths of the specific variables that can be streamed towards a Collector (in a similar way one can explore OIDs using a MIB browser). Installation is relatively straightforward as explained in the instructions given on the GitHub page of the tool. The user interface is similar to the user interface of MIB browsers: it contains the set of supported variables, their description and xpaths. Figure A.1 shows the key sections of the tool:

- An Explorer on the left-hand side, where YANG Models can be added to test their availability and operation and explore their content.
- Device Settings in the middle where the parameters of the connection towards the router can be set up. Also, in the Manage Models tab it is possible to add/compile new YANG models to the Explorer section up to all those that the router supports.
- An Encoding section in the middle where YANG Explorer creates NetConf RPC, Python or YDK, or RestConf RPC code for the dial-in requests, and an end Console section where the results of the requests are displayed.
- A Property section on the right-hand side which shows the properties of the selected YANG sections or variables (in this case capabilities requested for the router), and their xpath filter.



Explorer	Values	Operation
CISCO-IPSLA-AUTOMEASURE-MIB		
CISCO-IPSLA-ECHO-MIB		
CISCO-IPSLA-JITTER-MIB		
Cisco-IOS-XE-ip-sla-oper		
Cisco-IOS-XE-native		
SNMP-FRAMEWORK-MIB		
ietf-interfaces		
ietf-netconf-monitoring		
netconf-state		
capabilities		
capability	<get>	
datastores		
schemas		
sessions		
statistics		
get-schema		

Profile: XE1
Platform: csr
Host: 172.16.0.82 Port: 830
Username: [redacted] Password: [redacted]
[✓ NetConf] RestConf

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter>
      <netconf-state xmlns="urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring">
        <capabilities>
          <capability/>
        </capabilities>
      </netconf-state>
    </filter>
  </get>
</rpc>
```

Property	Value
Name	capability
Node Type	leaf-list
Data Type	inet:uri
Access	read-only
Presence	
Key	
Mandatory	
Default	
Path	ietf-netconf-monitoring/ netconf-state/capabilities/ capability
Description	List of NETCONF capabilities supported by the server.None
XPath Filter	/ncm:netconf-state/ capabilities/capability

Figure A.1: YANG Explorer screen

Upon executing (Run button) the RPC message, the router returns the appropriate set of YANG-NETCONF capabilities.

Similarly, it is possible to create a dial-in request for any variable or a branch in the YANG model trees. Figure A.2 shows the RPC message that requests an interface description on a router.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter>
      <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
        <interface>
          <description/>
        </interface>
      </interfaces>
    </filter>
  </get>
</rpc>
```

Figure A.2: RPC message that requests an interface description on a router

Figure A.3 shows the returned response which includes description fields for all active interfaces.

```
<rpc-reply message-id="urn:uuid:bd05d6fd-2994-42fd-8cdd-1808f474f129"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
      <interface>
        <description>MANAGEMENT</description>
      </interface>
      <interface>
        <description>H5</description>
      </interface>
      <interface>
        <description>to_XE2</description>
      </interface>
    </interfaces>
  </data>
</rpc-reply>
```

Figure A.3: Description fields of all active interfaces in the response

YANG explorer can also prepare RestConf requests, such as the one shown in Figure A.4, which asks for state data for all interfaces.

```
{
   "method": "GET",
   "url": "/restconf/api/running/interfaces-state",
   "params": {
      "Content-type": "application/vnd.yang.data+json",
      "Accept": "application/vnd.yang.data+json,
application/vnd.yang.errors+json"
   },
   "data": {}
}
```

Figure A.4: RestConf request for the status of all interfaces

It is also possible to get Python code that can be used to subscribe to the data stream for a specific set of variables as shown in Figure A.5.

```
"""
    Netconf python example by yang-explorer (https://github.com/CiscoDevNet/yang-
explorer)

    Installing python dependencies:
    > pip install lxml ncclient

    Running script: (save as example.py)
    > python example.py -a 172.16.0.82 -u USERNAME -p PASSWORD --port 830
"""

import lxml.etree as ET
from argparse import ArgumentParser
from ncclient import manager
from ncclient.operations import RPCError

payload = """
<filter xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <interfaces-state xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces"/>
</filter>
"""

if __name__ == '__main__':

    parser = ArgumentParser(description='Usage:')

    # script arguments
    parser.add_argument('-a', '--host', type=str, required=True,
                        help="Device IP address or Hostname")
    parser.add_argument('-u', '--username', type=str, required=True,
                        help="Device Username (netconf agent username)")
    parser.add_argument('-p', '--password', type=str, required=True,
                        help="Device Password (netconf agent password)")
    parser.add_argument('--port', type=int, default=830,
                        help="Netconf agent port")
    args = parser.parse_args()

    # connect to netconf agent
    with manager.connect(host=args.host,
                         port=args.port,
                         username=args.username,
                         password=args.password,
                         timeout=90,
                         hostkey_verify=False,
                         device_params={'name': 'csr'}) as m:

        # execute netconf operation
        try:
            response = m.get(payload).xml
            data = ET.fromstring(response)
        except RPCError as e:
            data = e._raw

        # beautify output
        print(ET.tostring(data, pretty_print=True))
```

Figure A.5: Python code used to subscribe to the data stream for a specific set of variables

Appendix B Preparing an OVA Image for Cisco Virtual Service Installation

Probably the most complex part of the virtual service installation, and certainly the least documented, is the preparation of an appropriate virtual service image (OVA file). This Appendix provides an example of the Ubuntu v18.04.5 LTS (bionic-server) virtual machine preparation, with instructions based on the Ubuntu 18.04 LTS image for another Ubuntu 18.04 LTS computer.

1. Install KVM.

```
sudo apt install qemu-kvm libvirt-bin virtinst bridge-utils cpu-checker
```

2. Install the uv-tools package, a project providing easy tools for easy use of Ubuntu cloud images.

```
sudo apt install uvtool
```

3. Download the desired image (in our case Ubuntu 18.04 - bionic).

```
uvt-simplestreams-libvirt sync arch=amd64
uvt-simplestreams-libvirt --verbose sync --source http://cloud-
images.Ubuntu.com/daily release=bionic arch=amd64
uvt-simplestreams-libvirt query
```

4. Create SSH keys and run the new VM. In this example we are creating one VM with 1024MB RAM, one virtual CPU and max 40GB disk. The default user on the machine is Ubuntu, and we define the new password cisco.

```
ssh-keygen
uvt-kvm create bionic-server release=bionic arch=amd64 --memory 1024 --cpu 1 --
disk 40 --ssh-public-key-file ~/.ssh/id_rsa.pub --password cisco
```

5. Run the machine in VM manager and log in to it, either through the VM manager or via SSH (you will need to get the IP address first).

6. Install the desired software (in our case the perfSONAR Tools bundle) and configure the VM console connection which is required for the connection from the router to the virtual service. For console connection, create (as the superuser) the file /etc/init/ttys0:

```
# ttyS0 - getty
#
# This service maintains a getty on ttyS0 from the point the system is
# started until it is shut down again.

start on stopped rc RUNLEVEL=[12345]
stop on runlevel [!12345]

respawn
exec /sbin/getty -L 9600 ttyS0 vt102
```

7. Start ttyS0 with sudo start ttys0 and/or reboot the VM. After the reboot, a process like the one below should be shown:

```
   783 ttyS0    Ss+    0:00 /sbin/agetty -o -p -- \u --keep-baud 115200,38400,9600
  ttyS0 vt220
```

8. Shut down the VM.
9. Create a directory where the Ubuntu image is going to be built (e.g. bionic-server.)
10. In that directory, convert the qcow file into qcow2 using the command below. In this example the image of the previously prepared server was in /var/lib/uvtool/libvirt/images/, but on some other systems the folder might differ.

```
sudo qemu-img convert -p -c -o compat=0.10 -O qcow2
/var/lib/uvtool/libvirt/images/bionic-server.qcow ./bionic-server.qcow2
```

11. In the same directory, create the package.yaml file:

```
manifest-version: 1.0

info:
  name: Ubuntu
  description: "KVM Ubuntu 18.04 LTS"
  version: 1.1

app:
  # Indicate app type (vm, paas, lxc etc.,)
  apptype: vm

  resources:
   cpu: 10
   memory: 1024000
   vcpu: 1

   disk:
    - target-dev: hdc
      file: bionic-server.qcow2

   interfaces:
    - target-dev: net1

   serial:
    - console
    - aux

  # Specify runtime and startup
  startup:
    runtime: kvm
    boot-dev: hd
```

12. In the same directory, create a file version.ver which should match the version in the package.yaml file.

```
echo 1.0 > version.ver
```

13. Download the create_ova.sh file from here to the parent directory and give it exec privileges.
14. Run the script and wait for the OVA file to be finished:

```
./create_ova.sh -mts 200000 -mfs 100000 bionic-server
```

Copy the OVA file to the router where you want to install it (e.g. using SCP).

Glossary

CLI	Command Line Interface
CPU	Central Processing Unit
GBP	Google Protocol Buffers
IP	Internet Protocol
ISIS	Intermediate System to Intermediate System
JTI	Junos Telemetry Interface
KVM	Kernel-based Virtual Machine
LTS	Long Term Support
MDT	Model-Driven Telemetry
MIB	Management Information Base
NAT	Network Address Translation
OID	Object Identifier
OVA	Open Virtual Application
OWAMP	One-Way Active Measurement Protocol
PFE	Packet Forwarding Engine
QoS	Quality of Service
RE	Routing Engine
RPC	Remote Procedure Call
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SSH	Secure Shell
TWAMP	Two Way Active Measurement Protocol
UDP	User Datagram Protocol
VM	Virtual Machine