TechPost ELEVATED

DC – AI – Routing – Switching – Security – Wireless – Automation

Monitoring PFE Resources on EVO Routers

An HPE Juniper Networking Techpost

David Roy

2024-12-26

Contents

1 Monitoring PFE Resources on EVO Routers

David Roy - 12/26/2024

Explore another use case of the Utility MIB feature [1] in Junos and EVO. We previously discussed this feature in a separate Techpost [2] in the context of the SRX platform. Today, we'll focus on its application on the ACX7000 platform.

1.1 Introduction

Recently, we received a request from a customer who wanted to monitor specific PFE counters on the ACX 7K platform. As you may know, Juniper platforms are designed to provide transparency, allowing customers access to the embedded "shell" of the Packet Forwarding Engines.

The command they referred to was as follows:

```
droy@myacx> start shell pfe network fpc0
droy@myacx:pfe> show evo-pfemand resource usage
UNIT 0 MDB Profile: lean-edge
Resource Usage Capacity size
COS_HR_ELEM 100 1024 1
COS_VOQ 800 131072 1
COS_VOQ_CNCTR 800 196608 1
ECMP 3 32000 30
EEDB_1 0 49152 0
EEDB_2 0 49152 0
EEDB_3 0 65536 0
EEDB_4 0 65536 0
EEDB_5 0 32768 0
EEDB_6 0 32768 0
EEDB_7 2 32768 0
EEDB_8 105 32768 0
EGR_MC_GRP 0 262144 1
EGR_VSI 8 65536 30
ENCAP_EXT 0 1048576 1
ESEM 0 131072 30
EVPN_ESI_CLASS_ID0 512 1
FAILOVER 420 6000 30
FEC_1 8 78644 150
FEC_2 0 78644 150
FEC_3 9 104858 150
GLEM 0 64000 1
IFL_STATS 10 16000 1
ING_MC_GRP 5 262144 1
ING_VSI 8 43691 90
INLIF_1 104 65536 60
INLIF_2 0 65536 60
ISEM_1 12 98304 30
ISEM_2 0 131072 30
KAPS 79 2368000 0
LEM 2 655360 30
LEXEM_PMF 0 49152 1
MYMAC_TCAM 0 9 106
NONE 0 0 0
OAM_MEP_DB 8 24576 1
OAM_NON_ACC_DB 0 24576 1
TCAM_EPMF_80 24 0 1
TCAM_EXTRN_80 0 0 1
TCAM_IPMF1_80 972 0 1
TCAM_IPMF2_80 128 0 1
TCAM_IPMF3_80 6 0 1
TCAM_PMF_ALL_EXTRN_800 0 1
TCAM_PMF_ALL_INTRN_801130 51200 1
TCAM_TNL_TERM 0 16000 178
TRUNK_ID 1 255 1
VRRP_SEXEM_1 0 196608 30
```

While the traditional CLI provides extensive statistics for the control and data planes, there are situations where PFE commands—intended for deeper troubleshooting—offer valuable insights that might be worth

monitoring. So, how can this be achieved?

The most straightforward answer is to ask your Systems Engineer (SE) to file an Enhancement Request (ER) to make these statistics available through the standard CLI, and subsequently via Netconf RPC or Streaming Telemetry. Although this process can take some time, it's the proper approach to enhance the Junos and EVO observability toolkit.

But what if you need a solution right now, without waiting for a future software release? In this case, we're dealing with statistics only accessible at the PFE level. For statistics already exposed via CLI commands (RPC) but not yet available through Streaming Telemetry, we've addressed this gap by allowing customers to develop their own Telemetry Sensors [3]. Unfortunately, that feature doesn't apply to our current use case.

However, the Junos/EVO automation toolkit offers two alternative solutions. These one are particularly useful for monitoring and troubleshooting by integrating customized metrics into existing SNMP workflows.

1. **SNMP Script with Proprietary OID**: This method involves creating an SNMP script that uses a custom OID. The script is executed whenever the corresponding SNMP OID is queried. This approach works well for simple counters fetched via SNMP GET but isn't ideal for exposing a large number of counters (e.g., an SNMP table).
2. **Utility MIB with event policy**: This approach is quite similar but in this case an on-box script is run periodically, thanks to an event-policy and fill what we called the Juniper Utility MIB by which the root-OID is .1.3.6.1.4.1.2636.3.47.1.1.2.1. Objects stored in this MIB as dot-decimal key and supported values are today: counter64, integer, unsigned integer and string.

1.2 A Word About Dot-Decimal Notation

In SNMP Utility MIB, **instance keys** are encoded as dot-decimal values appended to the base OID to uniquely identify rows or entries within a table or specific metrics. This encoding follows a structured approach to ensure that the **instance** corresponds to meaningful keys.

Here's a step-by-step explanation of how instances are encoded:

1. **Base OID**: The base OID defines the general MIB object (e.g., a table or metric). For Juniper Utility MIB, the root OID is .1.3.6.1.4.1.2636.3.47.1. But depending on the type of the data you want to fill in, the base OID will slightly change, i.e if you want to store counter64 values, the base OID will be: .1.3.6.1.4.1.2636.3.47.1.1.2.1.2

The following table lists all the base OID:

MIB Object	Base OID
jnxUtil	.1.3.6.1.4.1.2636.3.47.1
jnxUtilData	.1.3.6.1.4.1.2636.3.47.1.1
jnxUtilCounter32Table	.1.3.6.1.4.1.2636.3.47.1.1.1
jnxUtilCounter32Entry	.1.3.6.1.4.1.2636.3.47.1.1.1.1
jnxUtilCounter32Name	.1.3.6.1.4.1.2636.3.47.1.1.1.1.1
jnxUtilCounter32Value	.1.3.6.1.4.1.2636.3.47.1.1.1.1.2

MIB Object	Base OID
jnxUtilCounter32Time	.1.3.6.1.4.1.2636.3.47.1.1.1.1.3
jnxUtilCounter64Table	.1.3.6.1.4.1.2636.3.47.1.1.2
jnxUtilCounter64Entry	.1.3.6.1.4.1.2636.3.47.1.1.2.1
jnxUtilCounter64Name	.1.3.6.1.4.1.2636.3.47.1.1.2.1.1
jnxUtilCounter64Value	.1.3.6.1.4.1.2636.3.47.1.1.2.1.2
jnxUtilCounter64Time	.1.3.6.1.4.1.2636.3.47.1.1.2.1.3
jnxUtilIntegerTable	.1.3.6.1.4.1.2636.3.47.1.1.3
jnxUtilIntegerEntry	.1.3.6.1.4.1.2636.3.47.1.1.3.1
jnxUtilIntegerName	.1.3.6.1.4.1.2636.3.47.1.1.3.1.1
jnxUtilIntegerValue	.1.3.6.1.4.1.2636.3.47.1.1.3.1.2
jnxUtilIntegerTime	.1.3.6.1.4.1.2636.3.47.1.1.3.1.3
jnxUtilUintTable	.1.3.6.1.4.1.2636.3.47.1.1.4
jnxUtilUintEntry	.1.3.6.1.4.1.2636.3.47.1.1.4.1
jnxUtilUintName	.1.3.6.1.4.1.2636.3.47.1.1.4.1.1
jnxUtilUintValue	.1.3.6.1.4.1.2636.3.47.1.1.4.1.2
jnxUtilUintTime	.1.3.6.1.4.1.2636.3.47.1.1.4.1.3
jnxUtilStringTable	.1.3.6.1.4.1.2636.3.47.1.1.5
jnxUtilStringEntry	.1.3.6.1.4.1.2636.3.47.1.1.5.1
jnxUtilStringName	.1.3.6.1.4.1.2636.3.47.1.1.5.1.1
jnxUtilStringValue	.1.3.6.1.4.1.2636.3.47.1.1.5.1.2
jnxUtilStringTime	.1.3.6.1.4.1.2636.3.47.1.1.5.1.3

2. **Instance Key Values**: Each row or entry in a table may have one or more keys (e.g., indexes) that identify it. These keys can be integers, strings, or IP addresses.
3. Encoding the Instance:
    - **Integer Keys**: Encoded directly as the integer value. For example, key 5 becomes .5.
    - **String Keys**: Encoded as a length prefix followed by ASCII codes of each character. For instance, the string COS_VOQ is encoded as 67.79.83.95.86.79.81 (ASCII values of C, O, S, _, V, O, and Q).
    - **IP Address Keys**: Encoded as four dot-separated integers. For example, 192.168.1.1 becomes .192.168.1.1.
4. **Combining Base OID and Instance**: The base OID is concatenated with the encoded instance to form the complete unique OID.

For example:

- Base OID: .1.3.6.1.4.1.2636.3.47.1.1.2.1.2 (we want to store a Counter64 value – refer to the previous table)

- Instance name: COS_VOQ
- Resulting OID: .1.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.86.79.81

This encoding ensures that each entry or metric within the MIB hierarchy is uniquely identified, enabling accurate querying and monitoring.

1.3 Step-by-Step Process to Address Our Use Case

1.3.1 Step 1: Analyze the PFE Command

We identified the PFE command to parse: show evo-pfemand resource usage. The detailed output has been shown above. This command generates raw data; for every "resource" there are three counters: usage, capacity, and size. Thus, for each "resource," three OIDs will be created.

We will use the following rule to create a unique "instance" key:

`<RESOURCE-NAME>_[usage|capacity|size]`

For instance, the EEDB_3 resource will have three instances:

- `EEDB_3_usage`, represented in dot-decimal format as `69.69.68.66.95.51.95.117.115.97.103.101`
- `EEDB_3_capacity`, represented as `69.69.68.66.95.51.95.99.97.112.97.99.105.116.121`
- `EEDB_3_size`, represented as `69.69.68.66.95.51.95.115.105.122.101`

The instance name encoded as dot-decimal will then be concatenated with the base OID.

1.3.2 Step 2: Develop the On-Box Python Script

We chose Python 3 to write the on-box script, leveraging the PyEZ embedded library to simplify device interactions. The complete code, including detailed comments for each step, is shown below. Here's a summary of the approach:

- We used the Device and StartShell packages to open an internal RE shell on the device and execute the PFE command.
- The cprod utility was employed from the RE shell to issue the PFE command (specifically on FPC0).
- Finally, we used the request_snmp_utility_mib_set RPC to populate the Utility MIB

```
from jnpr.junos import Device
from jnpr.junos.exception import *
from jnpr.junos.utils.start_shell import StartShell
import jcs

def main():
    # CREATE THE LOCAL DEVICE OBJECT AND OPEN CONNECTION
    dev = Device()
    dev.open()
    try:
        # OPEN A SHELL FROM THE RE AND ISSUE THE PFE COMMAND BY USING CPROD UTILITY
        with StartShell(dev) as sh:
            op = sh.run('cprod -A fpc0 -c "show evo-pfemand resource usage"')
            # RETRIEVE THE OUTPUT AND CREATE A LIST OF LINES
            lines = op[1].strip().split("\n")

            # REMOVE EXTRA SPACE CHARACTER AND CREATE ROWS FOR EACH LINE
            # SO EACH LINE BECOMES. EX:
            #
            # COS_HR_ELEM 105 1024 1
            # COS_VOQ 840 131072 1
```

```
            # COS_VOQ_CNCTR 840 196608 1
            #
            # ROW 0 IS THE RESSOURCE NAME
            # ROW 1 THE USAGE
            # ROW 2 THE CAPACITY
            # ROW 4 THE SIZE
            rows = [line.split() for line in lines[1:]]

            # INIT OUR DICTIONNARY
            # AND FOR EACH RESOURCE (ROW 0) WE CREATE A NESTED DICT
            # WITH 3 COUNTERS: USAGE, CAPACITY & SIZE
            result = {}
            for row in rows:
               try:
                  resource, usage, capacity, size = row[0], row[1], row[2], row[3]
                  result[resource] = {
                     "usage": int(usage),
                     "capacity": int(capacity),
                     "size": int(size),
                  }
               except:
                  pass

            # LOOP THROUGH THE RESULTS TO FILL IN THE UTILITY MIB
            # WE USE request_snmp_utility_mib_set RPC
            for resource_name, metrics in result.items():
               for metric, value in metrics.items():
                  dev.rpc.request_snmp_utility_mib_set(
                     # BASE OID WILL BE .1.3.6.1.4.1.2636.3.47.1.1.2.1.2 (COUNTER64)
                     object_type="counter64",
                     # INSTANCE NAME IS <RESOURCE_NAME>_[usage|capacity|size]
                     # AND WILL BE ENCODED IN DOT-DECIMAL
                     instance=resource_name+"_"+metric,
                     # THE VALUE MUST BE PASSED AS A STRING AND WILL BE
                     # FURTHER ENCODED AS COUNTER64
                     object_value=str(value),
                  )

      except Exception as err:
         jcs.trace("ERROR", f" Unexpected error: {str(err)}")
         dev.close()
         return

      jcs.trace("INFO", " UTILITY Script successfully run")
      dev.close()

if __name__ == "__main__":
   main()
```

1.3.3 Step 3: Deploy the Python Script on the Router

First, you need to upload the script utility-evo.py to the following directory: /var/db/scripts/event.

Next, grant execution permissions to the file by running the following command:

```
droy@myacx> file change-permission /var/db/scripts/event/utility-evo.py permission +x
```

If your platform has multiple routing engines, ensure that the script is uploaded to both Routing Engines. You can either do this manually or allow the system to synchronize the scripts between both routing engines (this approach is highly recommended).

To enable this automatic synchronization, simply commit the following configuration knob:

```
edit exclusive
set system scripts synchronize
commit and-quit
```

1.3.4 Step 4: Schedule the Script to Update the Utility MIB Periodically

To ensure the script runs periodically (every 5 minutes in our case), we use an event policy. This policy monitors a generated event called five_minutes and triggers our script accordingly. With this setup, the Utility MIB will be updated every 5 minutes. The first step is to create a “periodic and generated event”:

```
edit exclusive
set event-options generate-event five_minutes time-interval 300
commit and-quit
```

```
droy@myacx> file checksum sha-256 /var/db/scripts/event/utility-evo.py
SHA256 (/var/db/scripts/event/utility-evo.py) =
    dbb0eb379da921ce278c17f1e7d286d95e9f021c20fdc429090f7c6531e6dab4
```

The reason for generating the SHA-256 hash is to inform the JUNOS/EVO system to execute the script only if its hash matches the specified value. This ensures that the script hasn’t been modified by unauthorized users.

Next, create a policy to monitor the event and trigger the utility-evo.py script:

```
edit exclusive
set event-options policy evo-resource events five_minutes
set event-options policy evo-resource then event-script utility-evo.py
set event-options event-script file utility-evo.py python-script-user py-user
set event-options event-script file utility-evo.py checksum sha-256
    dbb0eb379da921ce278c17f1e7d286d95e9f021c20fdc429090f7c6531e6dab4
commit and-quit
```

1.3.5 Step 5: Troubleshoot Script Execution

On the ACX7K, script execution is monitored using the EVO application’s tracing toolkit. Logs and traces for our feature are handled by the cscript application. To view them, simply run the following CLI command:

```
droy@myacx> show trace application cscript
[...]
2024-12-24 10:02:18.229501236 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg =
     "event script processing begins"
2024-12-24 10:02:18.229548005 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: /usr/libexec/ui/cscript"
2024-12-24 10:02:18.229551132 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: -m"
2024-12-24 10:02:18.229551984 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: event"
2024-12-24 10:02:18.229552719 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: -p"
2024-12-24 10:02:18.229553349 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: /"
2024-12-24 10:02:18.229554025 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: -E"
2024-12-24 10:02:18.229554674 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: user root"
2024-12-24 10:02:18.229555337 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: -f"
2024-12-24 10:02:18.229558576 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: utility-evo.py"
2024-12-24 10:02:18.229559247 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: -Q3"
2024-12-24 10:02:18.229559883 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: -2"
2024-12-24 10:02:18.229560522 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: dbb0eb379da921ce278c17f1e7d286d95e9f021c20fdc429090f7c6531e6dab4"
2024-12-24 10:02:18.229561264 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: -R"
2024-12-24 10:02:18.229561936 re0:cscript:21792:TRACE_INFO CSCRIPT_ARGUMENTS_CSCRIPT_ARGUMENTS_1
     msg = "arg: lab"
2024-12-24 10:02:18.233925438 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg =
     "running event script 'utility-evo.py'"
2024-12-24 10:02:18.233931060 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg =
     "opening event script '/var/db/scripts/event/utility-evo.py'"
```

```
2024-12-24 10:02:18.233941997 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg = "reading event script 'utility-evo.py'"
2024-12-24 10:02:19.451067345 re0:cscript:21792:TRACE_INFO
2024-12-24 10:02:19.795164487 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg = "INFO UTILITY Script successfully run"
2024-12-24 10:02:19.863738054 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg = "event script execution successful for 'utility-evo.py' with return: 0"
2024-12-24 10:02:19.863789351 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg = "finished event script 'utility-evo.py'"
2024-12-24 10:02:19.863846175 re0:cscript:21792:TRACE_INFO CSCRIPT_EVENTS_CSCRIPT_EVENTS_1 msg = "event script processing ends"
```

1.3.6 Step 6: Collect Your New Counters

The final step is the most rewarding: collect and visualize your new counters. Start by using the following CLI command to view the SNMP counters directly on the box. The “ascii” option is used to convert the dot-decimal instances into more readable MIB objects.

```
droy@myacx> show snmp mib walk jnxUtil ascii

jnxUtilCounter64Value."COS_HR_ELEM_capacity" = 1024
jnxUtilCounter64Value."COS_HR_ELEM_size" = 1
jnxUtilCounter64Value."COS_HR_ELEM_usage" = 100
jnxUtilCounter64Value."COS_VOQ_CNCTR_capacity" = 196608
jnxUtilCounter64Value."COS_VOQ_CNCTR_size" = 1
jnxUtilCounter64Value."COS_VOQ_CNCTR_usage" = 800
jnxUtilCounter64Value."COS_VOQ_capacity" = 131072
jnxUtilCounter64Value."COS_VOQ_size" = 1
jnxUtilCounter64Value."COS_VOQ_usage" = 800
jnxUtilCounter64Value."ECMP_capacity" = 32000
jnxUtilCounter64Value."ECMP_size" = 30
jnxUtilCounter64Value."ECMP_usage" = 3
jnxUtilCounter64Value."EEDB_1_capacity" = 49152
jnxUtilCounter64Value."EEDB_1_size" = 0
jnxUtilCounter64Value."EEDB_1_usage" = 0
jnxUtilCounter64Value."EEDB_2_capacity" = 49152
jnxUtilCounter64Value."EEDB_2_size" = 0
jnxUtilCounter64Value."EEDB_2_usage" = 0
jnxUtilCounter64Value."EEDB_3_capacity" = 65536
jnxUtilCounter64Value."EEDB_3_size" = 0
```

You can now collect those data from a remote collector. Here, we walk/ traverse the entire SNMP JnxUtil MIB:

```
snmpwalk -v 2c -c public myAcx7k 1.3.6.1.4.1.2636.3.47.1.1.2.1

iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.72.82.95.69.76.69.77.95.99.97.112.97.99.105.116.121 = Counter64: 1024
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.72.82.95.69.76.69.77.95.115.105.122.101 = Counter64: 1
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.72.82.95.69.76.69.77.95.117.115.97.103.101 = Counter64: 100
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.86.79.81.95.67.78.67.84.82.95.99.97.112.97.99.105.116.121 = Counter64: 196608
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.86.79.81.95.67.78.67.84.82.95.115.105.122.101 = Counter64: 1
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.86.79.81.95.67.78.67.84.82.95.117.115.97.103.101 = Counter64: 800
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.86.79.81.95.99.97.112.97.99.105.116.121 = Counter64: 131072
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.86.79.81.95.115.105.122.101 = Counter64: 1
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.67.79.83.95.86.79.81.95.117.115.97.103.101 = Counter64: 800
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.69.67.77.80.95.99.97.112.97.99.105.116.121 = Counter64: 32000
iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.69.67.77.80.95.115.105.122.101 = Counter64: 30
```

You can even request only a specific counter via a GET command. For instance, let’s collect the counter “EEDB_3_usage” – From ASIC to Decimal = 69.69.68.66.95.51.95.117.115.97.103.101

```
snmpget -v 2c -c public myAcx7k iso
    .3.6.1.4.1.2636.3.47.1.1.2.1.2.69.69.68.66.95.51.95.117.115.97.103.101

iso.3.6.1.4.1.2636.3.47.1.1.2.1.2.69.69.68.66.95.51.95.117.115.97.103.101 = Counter64: 10
```

1.4 Conclusion

In conclusion, leveraging the Utility MIB with event policies on Junos and EVO platforms offers a powerful method to monitor and troubleshoot Packet Forwarding Engine (PFE) statistics not yet exposed through standard API. While waiting for standard support through Netconf RPC or Streaming Telemetry may not be viable, the proposed solutions—such as SNMP scripts and Utility MIB updates—provide immediate alternatives. By automating the collection of resource counters and encoding them into SNMP MIB objects, users can integrate these insights into their existing workflows. This approach enables efficient monitoring and better visibility, ultimately enhancing operational control over the platform's performance.

1.5 Useful links

- [1] https://www.juniper.net/documentation/en_US/junos/topics/task/operational/security-snmp-best-practices-utility-mib-using.html
- [2] https://juniper.github.io/techposts/using-junos-snmp-utility-mib-on-juniper-srx/article
- [3] https://www.juniper.net/documentation/us/en/software/junos/interfaces-telemetry/netconf/topics/task/sensor-junos-telemetry-interface-configuring.html

1.6 Glossary

- ACX7K – ACX Series 7000 (a series of routers by Juniper)
- CLI – Command-Line Interface
- cprod – Command-line utility in Junos to execute commands on FPCs
- EVO – Juniper EVO
- FPC – Flexible PIC Concentrator
- jcs – Junos Control System (trace and logging utility in Junos)
- MIB – Management Information Base
- OID – Object Identifier
- PFE – Packet Forwarding Engine
- PyEZ – Python for Junos (a Python library for interacting with Junos devices)
- RE – Routing Engine
- RPC – Remote Procedure Call
- SE – Systems Engineer
- SHA – Secure Hash Algorithm
- SNMP – Simple Network Management Protocol
- SRX – Juniper SRX (a series of security devices)

DC – AI – Routing – Switching – Security – Wireless – Automation

HEWLETT PACKARD ENTERPRISE

HPE.com