NMIS8 Modelling Training

This content is also relevant to NMIS9, the model files are 100%, the main difference being the use of models-custom and models-default for storing models and NMIS8 plugins are not compatible with NMIS9, these need to be ported.

Table of Contents

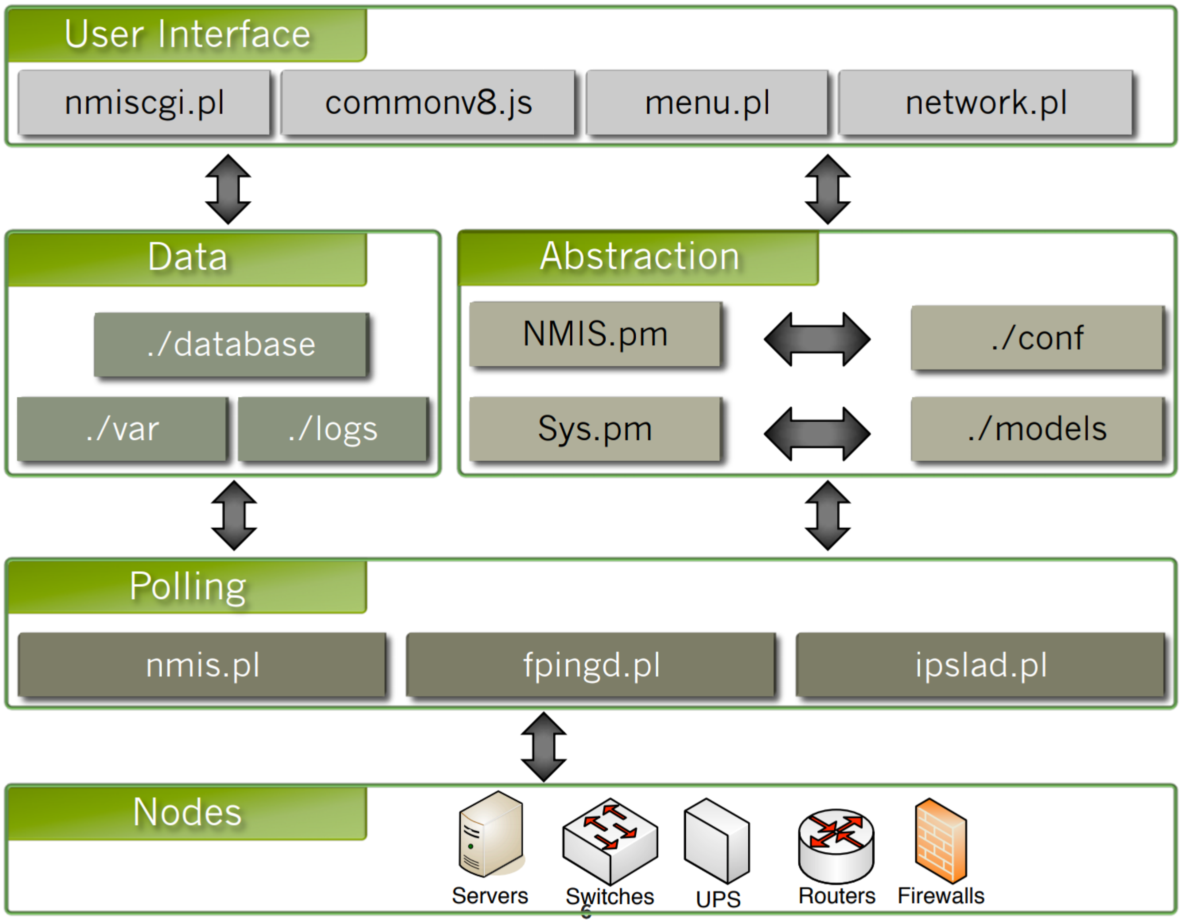
Basic Modelling

NMIS 8 Architecture

The nodes information is accessed using daemons or pollers that are part of the Polling System. The information obtained at this level could be ICMP (Internet Control Message Protocol), SNMP (Simple Network Management Protocol), etc.

Once the device information is collected, the Abstraction System is loaded, this system includes Sys.pm that specifies the methods to be able to communicate with the devices, loads information form the NMIS "VAR" directory and from the databases. The Abstraction System also includes NMIS.pm, which load NMIS and Devices configuration and is the responsible to load the models for the devices, collect information from the device base on that model, write down this information into the databases and into the VAR directory.

At the User Interface level, most of the heavy lifting is done by Network.pl, which loads up the Abstraction System and Data to produce the views, dashboard and everything displayed to the user in a meaningful and useful manner.



Modelling process

Types of Model Files

This are the files that we need to get familiar with in order to be able to successfully follow the modelling process.

Model.nmis : This file contains our model auto-discovery rules, it will determine, based on certain conditions, which model should be loaded.

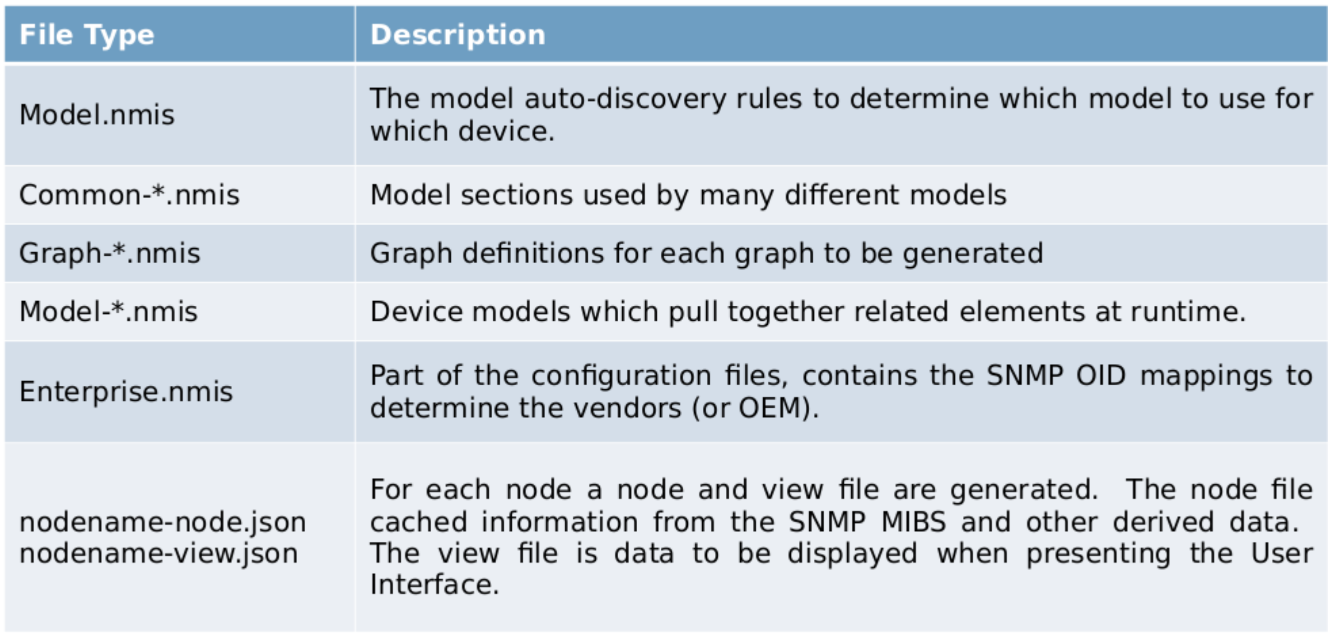
Common-*.nmis : A collection of files that contains common sections, these are included in many models that share certain features.

Graph-*.nmis : It contains the graph definitions for each graph to be generated and is loaded dynamically on runtime.

Model-*.nmis : Device models which pull together related elements obtained from the devices, it must include all the structure needed to represent the device on NMIS.

Enterprise.nmis : It is part of the config files, it lives on “/conf” directory and contains the SNMP OID mappings used by Model.nmis to match the vendors of the device.

nodename-node.json and **nodename-node.json** : For each node a node and view file are generated and maintained after a poll cycle (Update and Collect). It contains node cached information from the SNMP MIBS and other derived data, the data that it holds is temporarily persisted. The view file is data to be displayed when presenting the User Interface.

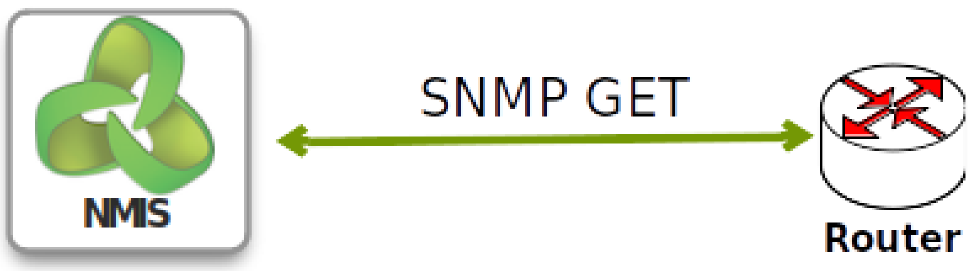


File Type	Description
Model.nmis	The model auto-discovery rules to determine which model to use for which device.
Common-*.nmis	Model sections used by many different models
Graph-*.nmis	Graph definitions for each graph to be generated
Model-*.nmis	Device models which pull together related elements at runtime.
Enterprise.nmis	Part of the configuration files, contains the SNMP OID mappings to determine the vendors (or OEM).
nodename-node.json nodename-view.json	For each node a node and view file are generated. The node file cached information from the SNMP MIBS and other derived data. The view file is data to be displayed when presenting the User Interface.

Overview

1. Collect From Device

The first step is to collect from the device sysDescr (System Description) and SysObjectId (Vendor's authoritative identification of the device). NMIS does a SNMP GET on the route and gets the sysDescr and SysObjectId.



From the sysObjectID, we obtain the ID and it is compared with the Enterprise.nmis file to match the vendor of the device. In this example the OID: 1.3.6.1.4.1.311 is a match for a Microsoft device.

From the sysDescr we obtain detailed information related to the device, in this case, the hardware and software description.

```
'sysDescr' => 'Hardware: Intel64 Family 6 Model 15 Stepping 6 AT/AT COMPATIBLE - Software: Windows Version 6.1
(Build 7600 Multiprocessor Free)'
'sysObjectID' => '1.3.6.1.4.1.311.1.1.3.1.1'
```

2. Determine the Vendor

Compare the sysObjectID to the Enterprises defined in Enterprise.nmis, the OID 311 will return the value of Enterprise, in this particular case, is 'Microsoft'.

conf/Enterprise.nmis

```
'311' => {
 'OID' => '311',
 'Enterprise' => 'Microsoft'
},
```

3. Auto-discovery model

To obtain the model to load, a matching process needs to be done on the Model.nmis file. The vendor name obtained from the previous step is used to find the section that matches its value, once we have the section to use, a regular expression will be perform to match the value of every item on that section against the sysDescr, this process will be done in numerical ascending order.
In this example, the vendor name will match the “Microsoft” section on the Model.nmis file, then, it will look for a match in order, starting from 10, it will try to identify if the value 'Windows Version 5.2' is included in the sysDescr. It will keep looking until a match is found. In this case, 'Windows Version 6.1' is a match, so the model to load is set to: 'Windows2008'.

models/Model.nmis

```
'Microsoft' => {
  'order' => {
    '30' => {
      'Windows2000' => 'Windows 2000 Version 5.0'
    },
    '10' => {
      'Windows2003' => 'Windows Version 5.2'
    },
    '20' => {
      'Windows2008' => 'Windows Version 6.1'
    }
  }
},
```

4. Load the model

The model result from Model.nmis will be loaded, the model filename must start with “Model-” followed by the name of the model as specified on the Model.nmis with extension “.nmis”.In our example, the model to use is 'Windows2008', so the file must be called: Model-Windows2008.nmis

```
$ ls -l /usr/local/nmis8/models/Model-Windows*
-rw-rw---- 1 nmis nmis 10920 Jul 10 14:51 Model-Windows2000.nmis
-rw-rw---- 1 nmis nmis  5761 Jul 10 14:51 Model-Windows2003.nmis
-rw-rw---- 1 nmis nmis  5964 Jul 10 14:51 Model-Windows2008.nmis
```

The model will load any common elements listed on the -common- section.

models/Model-Windows2008.nmis

```
'-common-' => {
  'class' => {
    'database' => {
      'common-model' => 'database'
    },
  --snip--
    'event' => {
      'common-model' => 'event'
    },
  }
},
```

Then the model will load specific sections listed after -common-.

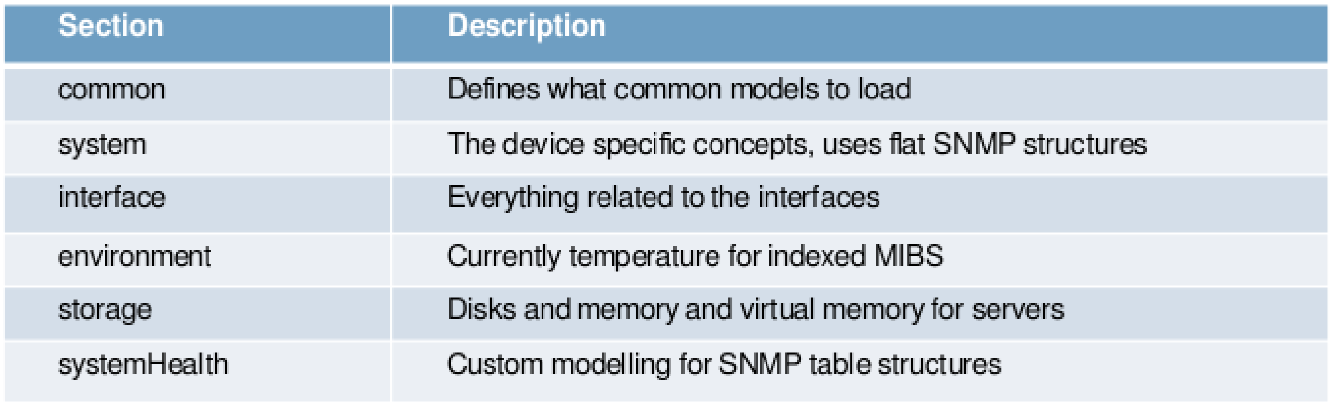
models/Model-Windows2008.nmis

```
'system' => {
  --snip--
},
'systemHealth' => {
  --snip--
},
'interface' => {
  --snip--
},
'device' => {
  --snip--
},
```

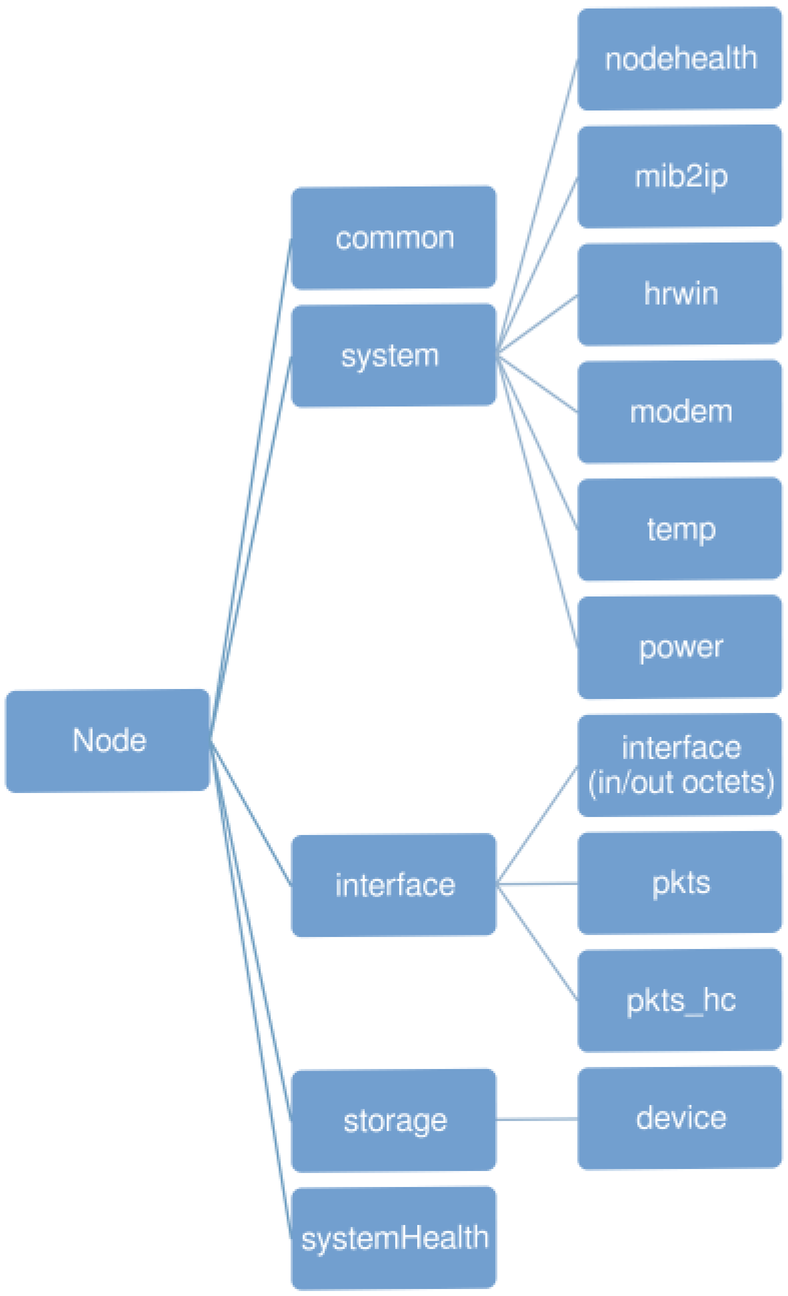
5. Load the data

Uses this model to collect device specific information from the device or to load the cached data from nmis8/var. If there is no file on the var directory for the device, NMIS will start collecting SNMP data from the device.
This process happens every time when dealing with the devices. When using the GUI, the model will be loaded with no SNMP enabled, however, when using NMIS.pl the model will be loaded with SNMP enabled.

Structure of a model

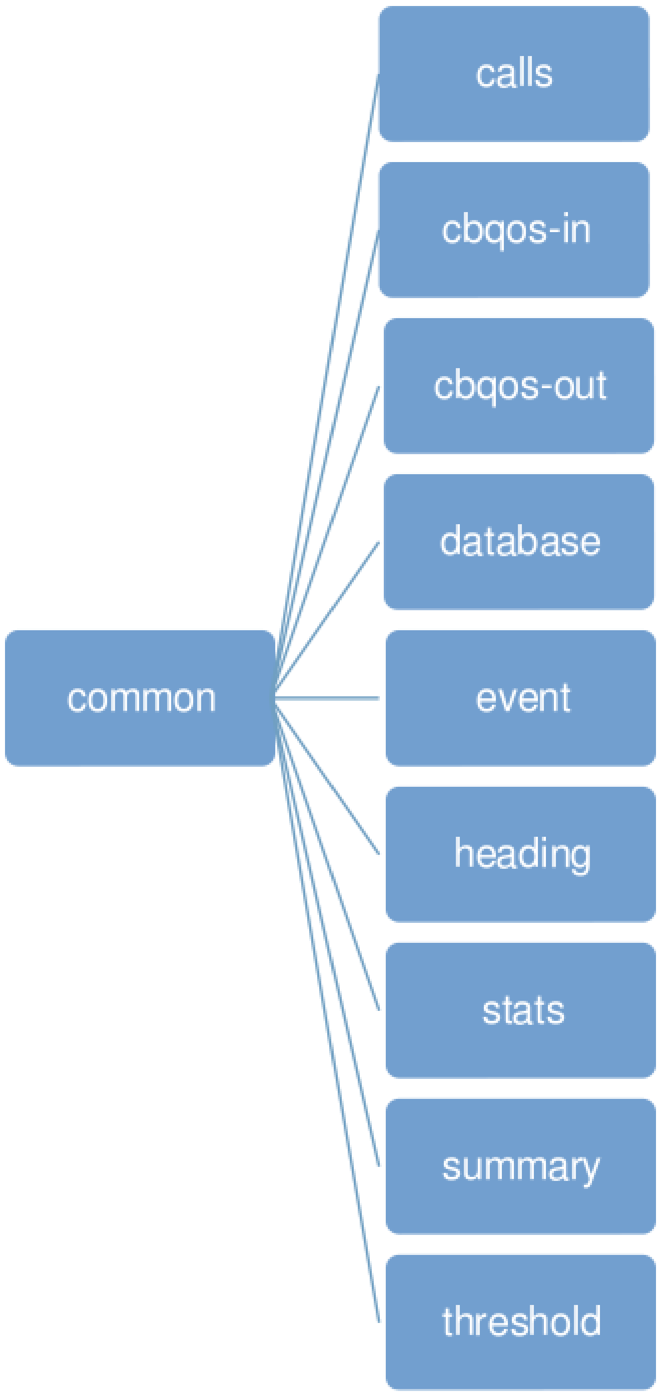


Section	Description
common	Defines what common models to load
system	The device specific concepts, uses flat SNMP structures
interface	Everything related to the interfaces
environment	Currently temperature for indexed MIBS
storage	Disks and memory and virtual memory for servers
systemHealth	Custom modelling for SNMP table structures



Common Models

Section	Description
calls	Monitoring for calls (less used today)
cbqos-in	Monitoring of the input traffic for Cisco Class Based QoS MIBS (HQoS)
cbqos-out	Monitoring of the output traffic for Cisco Class Based QoS MIBS (HQoS)
database	Definitions for the RRD database file names and folders
event	Event policy, determining what criticality what events are.
heading	Headings for various screens.
stats	RRD options for extracting stats from performance data.
summary	RRD options for generating summary data.
threshold	Thresholding policies for devices.



Modelling a Device

Goal for Modelling

What is the goal for the modelling? Just standard type support, or more advanced collection, do you want to collect some performance data about how a protocol is operating, or verify the number of sessions a firewall is running.

Sometimes you can find the source MIB in the documentation or a whitepaper, but sometimes it is very difficult to determine where the needed data is stored. If possible ask a product expert who is familiar with that specific product.

Device Instrumentation

Many times, people want to graph CPU and Memory, but not all devices support the collection of this information, you can only ask NMIS to collect something which the device has the instrumentation for, the MIBs should tell you if it is possible.

Relevance of Instrumentation

For Cisco routers, it is very handy to monitor CPU load, it is an excellent metric for how the device is performing, however on some newer Cisco devices, the processing is distributed and performed in hardware, so the CPU load is still handy but it may not be providing the information you need.

Verify the MIB Operation

Now you know what you want to collect and monitor, verify that the MIB operates the way the documentation says it does and verify that it works the way you think it does.

What do we need

Device Access

You can model a device without having one, but it is REALLY HARD, having SNMP readonly access to a device is vital for success.

SNMP MIBS

You will need to have all the necessary standard (IETF/IEEE) MIBs and the vendor specific MIBs for the device to be modelled.

SNMPWALK (SNMP Dump)

Once you have the MIBs the best way to interpret the MIBs is to complete an SNMP WALK of the device, first verify that you can use SNMP to access the device.

Then you need to do a full SNMP WALK, you will need to create a directory to store the MIBs in, for example ~/mibs, then copy the MIBs you obtained for the product and copy them to that folder, you will also need the standard MIBs, which are included in the NMIS distribution in /usr/local/nmis8/mibs/traps, copy these to the same folder, now verify that everything is working, you may get some errors from SNMP WALK about MIB compiling, but you can usually ignore those if you get a good output. For devices with proprietary MIB's or Enterprise MIBS, you should obtain them from the vendor, Google is very helpful and add them to your MIB's in ~ /mibs, before doing the SNMP walk.

Commands to run:

```
mkdir ~/mibs
cp <vendor mib file(s)> ~/mibs
cp /usr/local/nmis8/mibs/traps/* ~/mibs
snmpwalk -m ALL -M ~/mibs -v 2c -c GOODCOMMUNITY <HOSTNAME or IP ADDRESS> system
```

Expected output:

```
SNMPv2-MIB::sysDescr.0 = STRING: Hardware: Intel64 Family 6 Model 15 Stepping 6 AT/AT COMPATIBLE - Software:
Windows Version 6.1 (Build 7601 Multiprocessor Free)
SNMPv2-MIB::sysObjectID.0 = OID: SNMPv2-SMI::enterprises. 311.1.1.3.1.1
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (40604629) 4 days, 16:47:26.29
SNMPv2-MIB::sysContact.0 = STRING: dc_admin@opmantek.com
SNMPv2-MIB::sysName.0 = STRING: kaos
SNMPv2-MIB::sysLocation.0 = STRING: Head Office
SNMPv2-MIB::sysServices.0 = INTEGER: 79
```

Run a full walk and redirect to a file to obtain the SNMP Dump.

```
snmpwalk -m ALL -M ~/mibs -v 2c -c GOODCOMMUNITY > snmp_dump.txt
```

MIB Decoding Example

Let’s take this example, here snmpwalk couldn’t find a suitable MIB file to translate the SNMP data returned from the device. In this example, we are interested in the last three elements.

```
SNMPv2-SMI::enterprises.6302.2.1.1.1.0 = STRING: "Emerson Network Power”
SNMPv2-SMI::enterprises.6302.2.1.1.2.0 = STRING: "System Manager”
SNMPv2-SMI::enterprises.6302.2.1.1.3.0 = STRING: "1.6a 001”
SNMPv2-SMI::enterprises.6302.2.1.1.4.0 = "”
SNMPv2-SMI::enterprises.6302.2.1.2.1.0 = INTEGER: 6
SNMPv2-SMI::enterprises.6302.2.1.2.2.0 = INTEGER: 54014
SNMPv2-SMI::enterprises.6302.2.1.2.3.0 = INTEGER: 787097
SNMPv2-SMI::enterprises.6302.2.1.2.4.0 = INTEGER: 67
```

We know that “SNMPv2-SMI::enterprises” is always “.1.3.6.1.4.1” which makes these three MIBS:

- .1.3.6.1.4.1.6302.2.1.2.2.0
- .1.3.6.1.4.1.6302.2.1.2.3.0
- .1.3.6.1.4.1.6302.2.1.2.4.0

We want to find out what “.1.3.6.1.4.1.6302.2.1.2.2.0” represents, so we google it and no luck, let’s try with a more general OID, so we go up the tree a bit and grab “.1.3.6.1.4.1.6302” to find out to who this OID is assigned to. Google reveals that it belongs to “Emerson Energy Systems”, using our well developed google skills we were able to find the ees-power.mib file.

We want to figure out what each element is, so we have to trace the tree through the ASN.1[1] content of the MIB (Management Information Base).

SNMPv2-SMI::enterprises.6302.2.1.

```
ees OBJECT IDENTIFIER ::= { enterprises 6302 }
global OBJECT IDENTIFIER ::= { ees 2 }
powerMIB MODULE-IDENTITY
        LAST-UPDATED "200310140730Z"
        ORGANIZATION "
                Emerson Energy Systems (EES)"
        CONTACT-INFO "
                Emerson Energy Systems
                141 82 Stockholm
                Sweden"
        DESCRIPTION "
                Emerson Energy Systems (EES) Power MIB, revision B."
        ::= { global 1 }
ident OBJECT IDENTIFIER ::= { powerMIB 1 }
system OBJECT IDENTIFIER ::= { powerMIB 2 }
```

First we can notice that **enterprises.6302** is equivalent to **ees,** so we do a simple replacement, after this replacement we keep going down the MIB. The next element is **ees 2**, that becomes **global**, continuing with the same method, we figure out that **global 1** is **powerMIB**, **powerMIB 2** is **system** and finally **system 2** is **systemVoltage**.

SNMPv2-SMI::enterprises.6302.2.1.2 SNMPv2-SMI::enterprises.ees.global.powerMIB.system

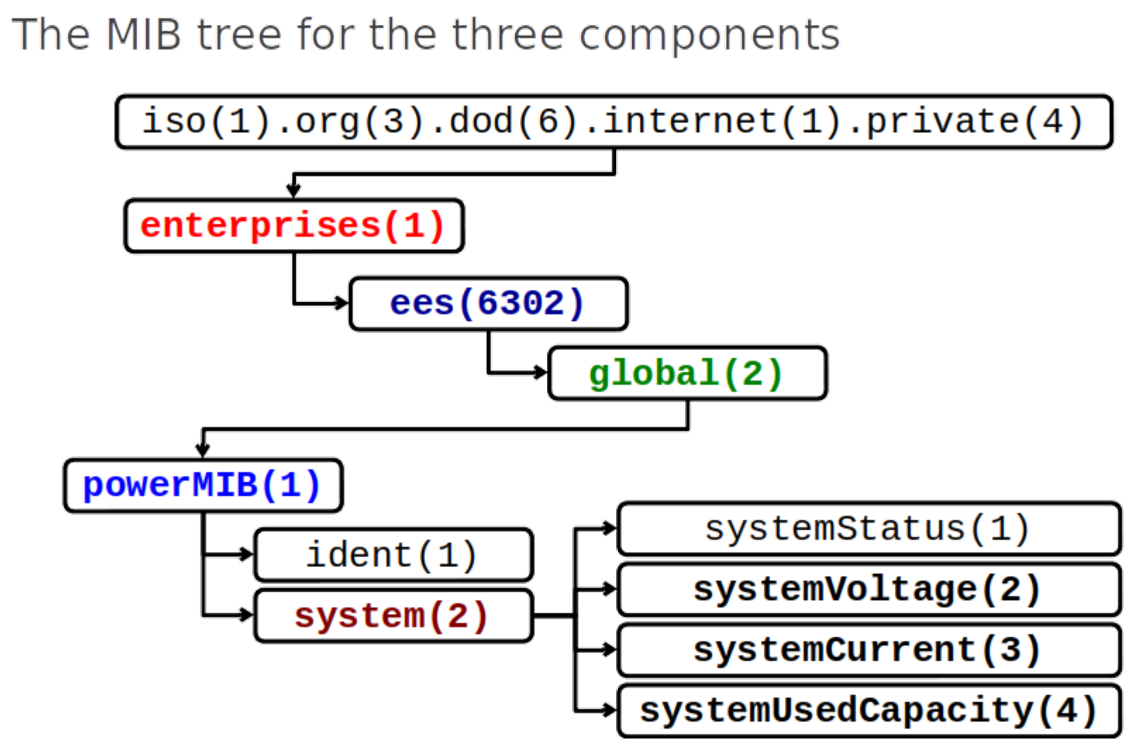
```
system OBJECT IDENTIFIER ::= { powerMIB 2 }
systemVoltage OBJECT-TYPE
        SYNTAX Integer32
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION "
                System voltage, stored as mV, including positive or negative
                sign. The integer 2147483647 represents invalid value."
        ::= { system 2 }

systemCurrent OBJECT-TYPE
        SYNTAX Integer32
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION "
                System current, stored as mA, including positive or negative
                sign. The integer 2147483647 represents invalid value."
        ::= { system 3 }

systemUsedCapacity OBJECT-TYPE
        SYNTAX Integer32
        MAX-ACCESS read-only
        STATUS current
        DESCRIPTION "
                Used capacity, stored as % of the total capacity.
                The integer 2147483647 represents invalid value."
        ::= { system 4 }
```

MIB tree for the three components

Here we have a graphical representation on how the MIB components are decomposed.



Implementing a new device model

As part of this training, we are going to implement our own model together, step by step, to have a better understanding on how the process is done. Now that we know how to decode a MIB and obtain the data we need to incorporate to the model using snmpwalk, let's add a new device model to NMIS.

To illustrate the process, let’s imagine that we just got a brand new Sun Microsystems Server and we need to create a model for it. We know that the vendor OID is: 1.3.6.1.4.1.42 (Sun), also, we ran an snmpwalk to the device and obtained the SysDescr and SysObjectId, with the following information:

```
SNMPv2-MIB::sysDescr.0 = STRING: SunOS opensolaris 5.11 snv_101b i86pc
SNMPv2-MIB::sysObjectID.0 = OID: UCD-SNMP-MIB::solaris
```

With this information we can start adding our model to NMIS. First, let’s add the vendor OID to the   /usr/local/nmis8/conf/Enterprise.nmis file.

conf/Enterprise.nmis

```
--snip--
 '42' => {
  'Enterprise' => 'Sun Microsystems',
  'OID' => '42'
},
--snip--
```

Then in /usr/local/nmis8/models/Model.nmis, we add the new model name to the vendor, if it doesn’t exist, we have to add the vendor too. In this case we are using a regular expression to match the information obtained from the SysDescr to the model name.

models/Model.nmis

```
--snip--
'Sun Microsystems' => {
  'order' => {
    '10' => {
      'SunSolaris' => 'sol|Sun SNMP|SunOS'
    }
  }
},
--snip--
```

It is important to highlight that the names used on these 3 files must match to successfully add a model to NMIS.

This is a simple process and applies for devices using their own SNMP Agent, however, many vendors now uses Net-SNMP as Agent and some considerations must be taken in count.

For example, we have now another Sun Microsystems device, snmpwalk returned this information:

```
SNMPv2-MIB::sysDescr.0 = STRING: SunOS gsmolames1 5.10 Generic_142900-13 sun4v
SNMPv2-MIB::sysObjectID.0 = OID: NET-SNMP-MIB::netSnmpAgentOIDs.3
```

Here the sysObjectID shows that the agent used by the device is NET-SNMP and the OID that it’s assigned to it is: 1.3.6.1.4.1.8072, once again, with this information we proceed to follow the process listed previously.

- Add the vendor

conf/Enterprise.nmis

```
--snip--
'8072' => {
  'Enterprise' => 'net-snmp',
  'OID' => '8072'
},
--snip--
```

- Update or add the information related to this vendor on Models.nmis

models/Model.nmis

```
--snip--
'net-snmp' => {
  'order' => {
    '10' => {
      'net-snmp' => 'Linux|SunOS|Darwin'
    }
  }
},
--snip--
```

With these changes, our model will be loaded for the device.

Tip: Many devices now uses NET-SNMP as Agent, most of the time, it’s a good idea to copy an existing model and tailor it to our needs instead of creating a new one.

Now you should be able to do it yourself, please refer to the Hands-On guide included with this material*****. (*****The mentioned material has not been created yet)

Adding a New Metric to Node Health

So far, we have been able to create our new model, now it is time to incorporate new metrics. It is very common and useful to display metrics on the Node Health. Let’s say that we have a Cisco Router and it is capable to provide the number of routes seen by a router, to keep things simple, we are going to edit an existing model for Cisco devices. As we know, a MIB dump is needed; it can be obtained running SNMPWALK against the device, as we are only interested now about the number of routes, we are going to focus on these 2 elements:

```
IP-FORWARD-MIB::ipCidrRouteNumber.0 = Gauge32: 33554432
IP-FORWARD-MIB::inetCidrRouteNumber.0 = Gauge32: 7
```

We may need to work with the OID’s for these MIBS, so we translate them and got:

- ipCidrRouteNumber = "1.3.6.1.2.1.4.24.3”
- inetCidrRouteNumber = "1.3.6.1.2.1.4.24.6"

Now that we have the needed information, it’s time to make some implementation decisions about names to use and OID’s that we will be using as data source. In this case, we have decided the following:

- We will call the new metric “routenumber”
- The MIB name is ipCidrRouteNumber
- The OID is 1.3.6.1.2.1.4.24.3
- The graph will be called routenumber

To add the metric to the node health successfully, we will need to make the following changes:

1. Add an entry to nmis_mibs.oid (Optional but handy)
2. Add it to the Model-CiscoRouter.nmis model to start collecting.
3. Add an entry to: Common-heading.nmis
4. Create a graph to view it: Graph-routenumber.nmis

Even though it is optional, we may want to map the OID to a MIB name; this could be done by adding it to /user/local/nmis8/mibs/nmis_mibs.oid.

mibs/nmis_mibs.oid

```
--snip--
"ipDefaultTTL"      "1.3.6.1.2.1.4.2"
"ipCidrRouteNumber" "1.3.6.1.2.1.4.24.3"
"inetCidrRouteNumber"   "1.3.6.1.2.1.4.24.6"
--snip--
```

It is recommended to do it this way, however it is no necessary but convenient because this mapping will be available for all the models as a MIB, so if we what to create other models there is no need to use the OID (**1.3.6.1.2.1.4.24.3**) to refer to it, we use the MIB name (**"ipCidrRouteNumber"**) instead.

Now, we add the metric to Model-CiscoRouter.nmis, We need to collect and store the new data into the RRD, to achieve that, we have to add a new variable called “RouteNumber” to “system -> rrd -> nodehealth -> snmp”, and specify that the values hold by this new variable are obtained from the MIB name “ipCidrRouteNumber” (OID: **1.3.6.1.2.1.4.24.3).** It is important to note that the maximum length for the variable name is 16 characters.

models/Model-CiscoRouter.nmis

```
--snip--
  'system' => {
    'nodeModel' => 'CiscoRouter',
    'nodeType' => 'router',
    'nodegraph' => 'health,response,cpu,mem-router,ip,frag,buffer,modem,calls',
    'rrd' => {
      'nodehealth' => {
        'threshold' => 'cpu,mem-proc',
        'graphtype' => 'buffer,cpu,mem-io,mem-proc,mem-router’ ,
        'snmp' => {
          'avgBusy5' => {
            'oid' => 'avgBusy5'
          },
          --snip--
          'RouteNumber' => {
              'oid' => 'ipCidrRouteNumber'
                }
        },
      },
--snip--
```

As mentioned before, mapping the OID is optional, we can point the values used by our new variable by specifying the OID. It is useful to use “snmpObject” to give a name as a note for other users to know what the OID is related to.

models/Model-CiscoRouter.nmis

```
--snip--
  'system' => {
    'nodeModel' => 'CiscoRouter',
    'nodeType' => 'router',
    'nodegraph' => 'health,response,cpu,mem-router,ip,frag,buffer,modem,calls',
    'rrd' => {
      'nodehealth' => {
        'threshold' => 'cpu,mem-proc',
        'graphtype' => 'buffer,cpu,mem-io,mem-proc,mem-router’ ,
        'snmp' => {
          'avgBusy5' => {
            'oid' => 'avgBusy5'
          },
          --snip--
          'RouteNumber' => {
                 'snmpObject' => 'ipCidrRouteNumber'
             'oid' => '1.3.6.1.2.1.4.24.3'
                }
        },
      },
--snip--
```

At this stage we also add the name of the graph, we have decided that our graph is going to be called “routenumber”, so we add it to the graphtype (“system -> rrd -> nodehealth -> graphtype”), now the model knows the name of the graph to use. In the same way, we need to tell the model that the graph has to be displayed, for that; we add the graph name to “nodegraph” (“system -> nodegraph”). The metrics will be shown keeping the same order specified on the “nodegraph”, in this case the graph will be displayed at the bottom, as we added it to the end of the list, we can change the order as needed.

models/Model-CiscoRouter.nmis

```
--snip--
  'system' => {
    'nodeModel' => 'CiscoRouter',
    'nodeType' => 'router',
    'nodegraph' => 'health,response,cpu,mem-router,ip,frag,buffer,modem,calls,routenumber',
    'rrd' => {
      'nodehealth' => {
        'threshold' => 'cpu,mem-proc',
        'graphtype' => 'buffer,cpu,mem-io,mem-proc,mem-router,routenumber’ ,
        'snmp' => {
          'avgBusy5' => {
            'oid' => 'avgBusy5'
          },
          --snip--
          'RouteNumber' => {
              'oid' => 'ipCidrRouteNumber'
                }
        },
      },
--snip--
```

Add the heading that the graph must show to: models/Common-heading.nmis

models/Common-heading.nmis

```
--snip--
 'heading' => {
   'graphtype' => {
      --snip--
      'routenumber' => 'Number of Routes’
   }
 }
--snip--
```

Now we have to create the graph, the filename must start with “Graph-“ followed by the exact name that was choosen for the graph, in this case “routenumber”, so the graph filename is: Graph-routenumber.nmis.

The graph file has different sections; each defines how the graph should be display. NMIS can display 2 sizes of graphs: “Standard” and “Small”, we can specify what details to show in either case.

Note that the “title” and “vlabel” (vertical label) uses “standard” and “short” by default to define the length of the item and the section “option” uses “standard” and “small” to define the type of graph, however, these variables can be change if needed.

models/Graph-routenumber.nmis

```
'title' => {
  'standard' => '$node - $length from $datestamp_start to $datestamp_end',
  'short' => '$node - $length'
},
'vlabel' => {
  'standard' => 'Number of Routes'
},
'option' => {
  'standard' => [
    'DEF:routes=$database:RouteNumber:AVERAGE',
    'LINE1:routes#0000ff:Number of Routes',
    'GPRINT:routes:AVERAGE:Avg Number of Routes %1.2lf',
    'GPRINT:routes:MAX:Max Number of Routes %1.2lf'
  ],
  'small' => [
    'DEF:routes=$database:RouteNumber:AVERAGE',
    'LINE1:routes#0000ff:Number of Routes',
    'GPRINT:routes:AVERAGE:Avg Number of Routes %1.2lf',
    'GPRINT:routes:MAX:Max Number of Routes %1.2lf'
  ]
}
```

Let's examine the first line inside "option -> standard" to understand what is the meaning of each term.

`DEF:`**`routes`**`=$database:`**`RouteNumber`**`:AVERAGE`

`DEF:<vname>=<rrdfile>:<ds-name>:<CF>`

vname: Internal variable where the data from the RRD will be stored, in our case: "routes"

rrdfile: The RRD where the data is stores, in our case: $database (already defined in Common-database.nmis)

ds-name: The Data Source name used to store particular data into the RRD, in our case "RouteNumber"

CF: The Consolidation Function can be AVERAGE, MINIMUM, MAXIMUM, or LAST, in our case: "AVERAGE"

The ds-name(RouteNumber) must be exactly the same as name given to the variable assigned in the model ("system -> rrd -> nodehealth -> snmp").

The next item defines the type of drawing that will be done with the data.

`LINE1:`**`routes`**`#0000ff:Number of Routes`

`LINE:value[#color]:[legend]`

LINE1: Type of drawing, in our case: a Line.

Value: The value or variable holding the value, in our case: routes.

#color: Hexadecimal color value, in our case: #0000ff (blue).

Legend: The legend associated to the value.

The last two lines are similar, are the calculated values displayed as part of the legend.

Basically, the graph is defined based on the RRD Tool graph system and additional information and extended options can be found here: https://oss.oetiker.ch/rrdtool/doc/rrdgraph.en.html

System Section

We have added a new metric to the nodeHealth section, but sometimes we need to add new concepts to the device model and create their own RRD files.

As we have did before, we need to take some implementations decisions:

- What data to collect?
- The MIB name is tcp
- Collection will be called tcp
- The graphs will be called "tcp-conn" and "tcp-segs"

To add the new collection to the system section successfully, we will need to make the following changes:

1. Add the “tcp” section to the system section of the model
2. Add an entry to: Common-heading.nmis.
3. Add an entry to: Common-database.nmis
4. Create a graph to view it: Graph-tcp-conn.nmis and Graph-tcp-segs.nmis
5. Add any required MIBS to nmis_mibs.oid (Optional)

We did an SNMPWALK and obtained the following information related to the tpc. It is important to notice that this MIBS is flat, there are only one value per MIB, which makes it possible to added to the system section.

```
RFC1213-MIB::tcpActiveOpens.0 = Counter32: 487232
RFC1213-MIB::tcpPassiveOpens.0 = Counter32: 110120
RFC1213-MIB::tcpAttemptFails.0 = Counter32: 99301
RFC1213-MIB::tcpEstabResets.0 = Counter32: 75577
RFC1213-MIB::tcpCurrEstab.0 = Gauge32: 72
RFC1213-MIB::tcpInSegs.0 = Counter32: 12879179
RFC1213-MIB::tcpOutSegs.0 = Counter32: 11516662
RFC1213-MIB::tcpRetransSegs.0 = Counter32: 428664
RFC1213-MIB::tcpInErrs.0 = Counter32: 6
RFC1213-MIB::tcpOutRsts.0 = Counter32: 69835
```

To illustrate better this process, we will be using an already existing model called: Model-net-snmp.nmis. Now we add the new section named “tcp” under “System -> rrd”.

Inside the “tcp” section, we add the graptype, listing the names of the graphs that we decide earlier. In addition, we have to add a new snmp section.

models/Model-net-snmp.nmis

```
  'system' => {
    --snip--
    'rrd' => {
      --snip—
      'tcp' => {
        'graphtype' => 'tcp-conn,tcp-segs',
        'snmp' => {
        }
      }
    }
  }
```

Now we add the OID to collect, inside the new snmp section, as mentioned previously, we can map the OID to a MIB name in the nmis_mibs.oid file or we can just use the numerical OID. In addition, we add the option item, which will indicate to the rrd if the data is a counter or a gauge, and the lower and upper limits, being “U” equivalent to Unlimited.

In order to figure out these details, we have to look at the SNMPWALK output, this will also help us to follow the names used on every OID. In our case the MIBS shows that the element “tcpCurrEstab” is a gauge.

models/Model-net-snmp.nmis

```
  'system' => {
    --snip--
    'rrd' => {
      --snip—
      'tcp' => {
        'graphtype' => 'tcp-conn,tcp-segs',
        'snmp' => {
          'tcpActiveOpens' => {
            'oid' => 'tcpActiveOpens',
            'option' => 'counter,0:U'
          },
          'tcpPassiveOpens' => {
            'oid' => 'tcpPassiveOpens',
            'option' => 'counter,0:U'
          },
          --snip—
          'tcpCurrEstab' => {
            'oid' => 'tcpCurrEstab',
            'option' => 'gauge,0:U'
          },
          --snip—
          'tcpOutRsts' => {
            'oid' => ‘1.3.6.1.2.1.6.15’,
            'snmpObject' => 'tcpOutRsts',
            'option' => 'counter,0:U'
          }
        }
      }
    }
  }
```

Next, we have to add the headings of our new graphs into Common-heading.nmis, these are the headings that will be shown when displaying graphs in NMIS. If not defined you will see a message like this: “heading not defined in Model”

models/Common-heading.nmis

```
--snip--
 'heading' => {
   'graphtype' => {
      --snip--
      'tcp-conn' => 'TCP Connections',
      'tcp-segs' => 'TCP Segments'
   }
 }
--snip--
```

Now we have to add a new entry to Common-database.nmis, it is important to match the name used on Common-database.nmis and the name of the new section.

models/Common-database.nmis

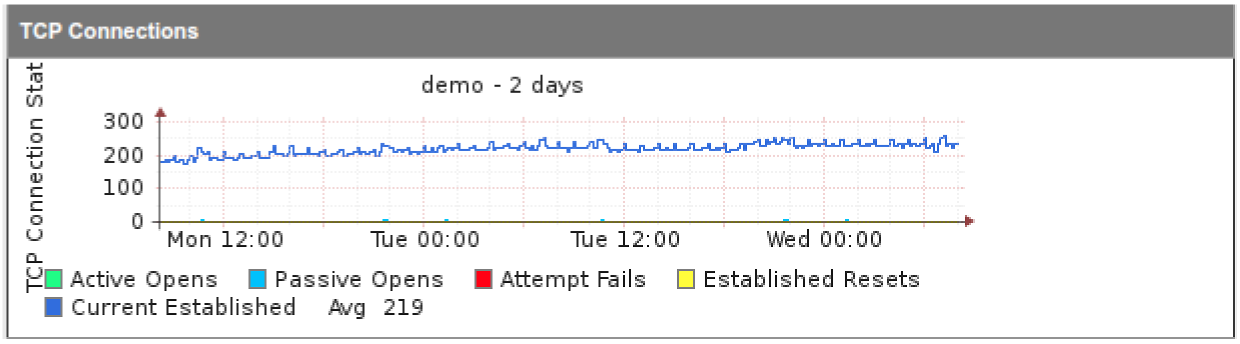
```
--snip--
'tcp' => '/nodes/$node/health/tcp.rrd',
--snip--
```

Our last step is to make meaningful graphs with the collected data. The same process we already discussed while adding a metric. In this case, more variables were added to the graph and areas and stacks are used instead of lines. Here is the implementation of Graph-tcp-conn.nmis.

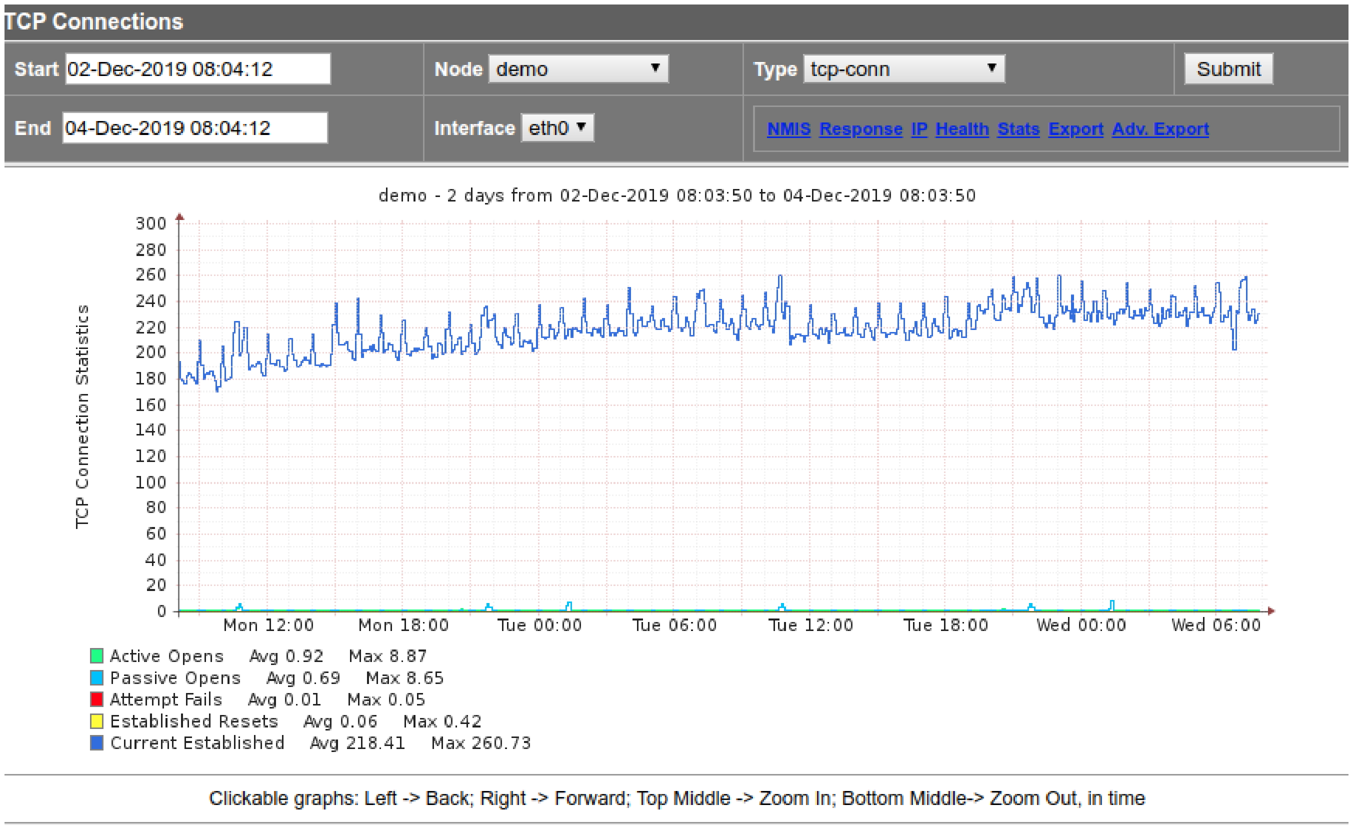
models/Common-heading.nmis

```
'title' => {
  'standard' => '$node - $length from $datestamp_start to $datestamp_end',
  'short' => '$node - $length'
},
'vlabel' => {
  'standard' => 'TCP Segment Statistics',
  'short' => 'TCP Segment Stats'
},
'option' => {
  --snip--
  'small' => [
    'DEF:tcpInSegs=$database:tcpInSegs:AVERAGE',
    'DEF:tcpInErrs=$database:tcpInErrs:AVERAGE’,
    'DEF:tcpOutSegs=$database:tcpOutSegs:AVERAGE',
    'DEF:tcpOutRsts=$database:tcpOutRsts:AVERAGE',
    'DEF:tcpRetransSegs=$database:tcpRetransSegs:AVERAGE’,
    'CDEF:tcpInSegsSplit=tcpInSegs,-1,*',
    'CDEF:tcpInErrsSplit=tcpInErrs,-1,*’,
    'AREA:tcpInSegsSplit#0000ff:Input Segments',
    'STACK:tcpInErrsSplit#ffff00:Input Errors’,
    'AREA:tcpOutSegs#00ff00:Output Segments',
    'STACK:tcpOutRsts#000000:Output Resets',
    'STACK:tcpRetransSegs#ff0000:Retransmitted',
  ]
}
```

Finally, the small version of the graph should look like this:



This is how the standard version looks:


TCP Connections

Start 02-Dec-2019 08:04:12 | Node demo | Type tcp-conn | Submit

End 04-Dec-2019 08:04:12 | Interface eth0 | NMIS Response IP Health Stats Export Adv. Export

demo - 2 days from 02-Dec-2019 08:03:50 to 04-Dec-2019 08:03:50

🟩	Active Opens	Avg 0.92	Max 8.87
🟦	Passive Opens	Avg 0.69	Max 8.65
🟥	Attempt Fails	Avg 0.01	Max 0.05
🟨	Established Resets	Avg 0.06	Max 0.42
🟦	Current Established	Avg 218.41	Max 260.73

Clickable graphs: Left -> Back; Right -> Forward; Top Middle -> Zoom In; Bottom Middle-> Zoom Out, in time

System Health Section (Indexes)

Often a section of data that is useful to have displayed in NMIS is presented in SNMP as a table. In order to model this NMIS modelling supports a "systemHealth" section that allows indexing to be used.

This time we need to add Disk IO information as a table to our model. The net-snmp model already has a section called "diskIOTable", however, we will be using its development as an example.

Once again, we need to take some implementations decisions:

- What table of data to collect?
- The MIB name is diskIOTable
- The collection will be called diskIOTable

We will need to make the following changes:

1. Add the systemHealth section to the model file
2. Add any required MIBS to nmis_oids.nmis (optional)
3. Add an entry to: Common-heading.nmis
4. Add an entry to: Common-database.nmis
5. Create any required graphs to view data: Graph-diskio-rw.nmis and Graph-diskio-rwbytes.nmis
6. Add new section diskIOTable to model_health_sections in Config.nmis

This is the information obtained with SNMPWALK, It is important to notice that this MIBS is indexed, which makes the use of tables on the systemhealth section possible.

```
UCD-DISKIO-MIB::diskIOIndex.26 = INTEGER: 26
UCD-DISKIO-MIB::diskIODevice.26 = STRING: sda
UCD-DISKIO-MIB::diskIONRead.26 = Counter32: 3524873216
UCD-DISKIO-MIB::diskIONWritten.26 = Counter32: 3281483776
UCD-DISKIO-MIB::diskIOReads.26 = Counter32: 1574933
UCD-DISKIO-MIB::diskIOWrites.26 = Counter32: 182695521
UCD-DISKIO-MIB::diskIONReadX.26 = Counter64: 7819840512
UCD-DISKIO-MIB::diskIONWrittenX.26 = Counter64: 1150037751808
```

It is a standard SNMP Table construct. For every disk that we got, there is a index assigned to it and every disk will have `“diskIOIndex, diskIODevice,diskIONRead,etc”.`

Model file

The systemHealth section is a "top-level" section, which means it does not sit inside another section.

```
%hash = (
-- SNIP --
  'systemHealth' => {
    'sections' => 'diskIOTable',
    'sys' => {
      'diskIOTable' => {
      'indexed' => 'diskIOIndex',
      'headers' => 'diskIODevice',
      'snmp' => {
        'diskIOIndex' => {
          'oid' => 'diskIOIndex',
          'title' => 'IO Device Index'
        },
        'diskIODevice' => {
          'oid' => 'diskIODevice',
          'title' => 'IO Device Name'
        },
        -- SNIP --
      }
    },
    'rrd' => {
      'diskIOTable' => {
        'control' => 'CVAR=diskIODevice;$CVAR =~ /sda|sr|disk|xvda|dm\-/',
        'indexed' => 'true',
        'graphtype' => 'diskio-rw,diskio-rwbytes',
        'snmp' => {
          -- SNIP --
        }
      }
    }
  },
-- SNIP --
);
```

At the top of the **systemHealth** section a "**sections**" key tells NMIS which sections to display in the GUI (along the top column of links while viewing the node). This name must match the name of the section below.

In this example `"diskIOTable"` is used.

sys section

In the above code snippet there is a '**sys**' section, this is where data that will be stored in the Nodename-node.nmis file is defined. This is also where data that is needed for gathering the RRD section is defined. If you want to see the latest value gathered by NMIS for these MIBS check the Nodename-node. nmis file for your node. The values defined inside the snmp section are like any other part of the model.

The sys->diskIOTable section specifies 2 keys:

1. `indexed => 'diskIOIndex'`
   This tells NMIS that the OID diskIOIndex will hold the index value to loop on.
2. `headers => 'diskIODevice'`
   This tells NMIS that when displaying the indexes, a column for diskIODevice should be shown. If the headers directive lists more than one key, then all the corresponding columns will be present; in the example above we could have included the diskIOIndex column, too (but the device name is more useful in this case).

3. `snmp => {}`
   This tells NMIS what data to collect.
   3.a `oid` It can be something nmis_mibs.oid or an OID.
   3.b `title` what to call it when is displayed.

NMIS versions before 8.4.8 require that you list the OID name-to-raw-oid association in mibs/nmis_mibs.oid; for the example above entries for diskIODevice and diskIOIndex would be required. To simplify the modelling process, 8.4.8 and later also support the key index_oid which lets you associate a raw oid with a name in one model file. Using that the example would look like this:

```
'sys' => {
  'diskIOTable' => {
   'indexed' => 'diskIOIndex',
   'index_oid' => '1.3.6.1.4.1.2021.13.15.1.1.1',
...
```

Another feature in 8.4.8 is index_regex, which allows multi-element indexing: normally SNMP tables are indexed by the last, single numeric OID component. When NMIS does an update on a indexed entity, it iterates through all the known values for this index component and records them. This iteration does not work if the index consists of more than one number, as it does on certain equipment. In such cases you can set index_regexto a value that captures the OID components that vary between table elements. For example,

```
'index_regex' => '\.(\d+\.\d+\.\d+)$',
```

ensures that the last three numbers are used for indexing.

rrd section

The rrd section defines what data will be collected and stored into rrd's. Once again, the values defined inside the snmp section are like any other part of the model.

The rrd->diskIOTable section specifies 3 keys:

1. `'control' => 'CVAR=diskIODevice;$CVAR =~ /sda|sr|disk|xvda|dm\-/',`
   This tells NMIS that the OID diskIODevice should be checked and only capture the values into RRD if they match the regular expression given.
2. `indexed => 'true',`
   Tell NMIS this is an indexed table, it will then go and use the index specified in the sys section above to iterate.

3. `graphtype => 'diskio-rw,diskio-rwbytes'`
   what graph-types will this rrd section create data for.

4. `snmp => 'diskio-rw,diskio-rwbytes'`
   what graph-types will this rrd section create data for.
   4.a `oid` can be something nmis_mibs.oid or an OID.
   4.b `option` is the data a counter or gauge, and what are the lower and upper limits.
      e.g: `'option' => 'counter,0:U'`
   4.c `title` what to call it when it displayed.

Common-heading.nmis file

These are the headings you will see when displaying the graph in various screens in NMIS. If not defined you will see a message like this: “heading not defined in Model”.

```
'diskio-rw' => 'Disk IO Blocks',
'diskio-rwbytes' => 'Disk Read Write Bytes'
```

Common-database.nmis file

The name of the rrd file is specified in this file. You will want a new set of files for your new section, to do that simply add a new line.

```
'diskIOTable' => '/health/$nodeType/$node-diskiotable-$index.rrd',
```

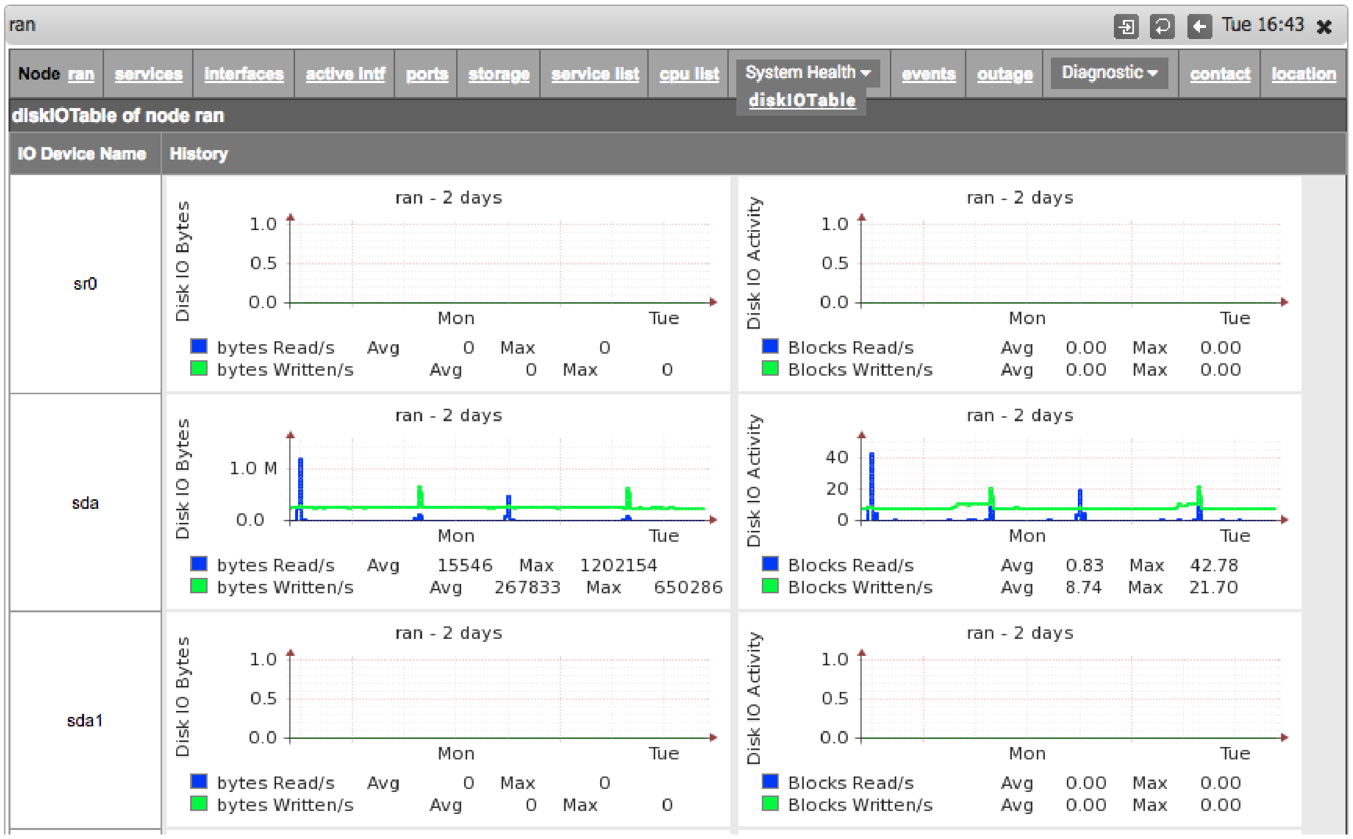
As you can see, the file name has **$index** in the name so NMIS will create a new file for each index that it is gathering.

Graph-diskio-rw.nmis and Graph-diskio-rwbytes.nmis files

These are the files used to define the graphs. We define the RRD "DEF" based on what you stored, defined the LINE or AREA to graph, use some GPRINTS for text output and other RRD syntax to achieve the desired graph.

```
'title' => {
  'standard' => '$node - $length from $datestamp_start to $datestamp_end',
  'short' => '$node - $length'
},
'vlabel' => {
  'standard' => 'Disk IO Activity'
},
'option' => {
  'standard' => [
    'DEF:diskIOReads=$database:diskIOReads:AVERAGE',
    'DEF:diskIOWrites=$database:diskIOWrites:AVERAGE',
    'LINE2:diskIOReads#0000ff:Blocks Read/s\t',
    'GPRINT:diskIOReads:AVERAGE:Avg %8.2lf',
    'GPRINT:diskIOReads:MAX:Max %8.2lf\\n',
    'LINE2:diskIOWrites#00ff00:Blocks Written/s\t',
    'GPRINT:diskIOWrites:AVERAGE:Avg %8.2lf',
    'GPRINT:diskIOWrites:MAX:Max %8.2lf\\n',
  ]
}
```

Once all this procedure has been done, the new sections will appear under the "System Health" Drop-down menu.



```
sr0
bytes Read/s     Avg        0   Max         0
bytes Written/s       Avg        0   Max         0
Blocks Read/s          Avg   0.00   Max    0.00
Blocks Written/s       Avg   0.00   Max    0.00

sda
bytes Read/s     Avg    15546   Max   1202154
bytes Written/s       Avg   267833   Max    650286
Blocks Read/s          Avg   0.83   Max   42.78
Blocks Written/s       Avg   8.74   Max   21.70

sda1
bytes Read/s     Avg        0   Max         0
bytes Written/s       Avg        0   Max         0
Blocks Read/s          Avg   0.00   Max    0.00
Blocks Written/s       Avg   0.00   Max    0.00
```

Thresholding

NMIS8 includes powerful capabilities for performance and operational thresholding, which greatly enhance network management capabilities. These thresholds result in alerts/events/notifications which NMIS can send when it sees a threshold breached. The thresholds have very granular controls which by default have been configured fairly broadly.

Considerations

Thresholding can be accomplished with the following steps:

- What data is being collected which can be thresholded?
- Add a threshold property to the model section.
- Add threshold values to the Common-threshold.nmis file.
- Add statistics extraction (pulling data from the RRD file and formating it) to the Common-stats.nmis file.
- Test the newly provisioned thresholding policy.
- Consider advanced threshold using controls (Adding boolean logic and regular expressions for precision selection).

Consider the following questions:

- What would you like to threshold?
- How feasible is the thresholding candidate?
- Can the metrics be reduced/translated/combined into a meaningful threshold?
- What should the corresponding event name for the threshold be?
- The event name must include "Proactive" at the beginning in order for NMIS to process it correctly. e.g. "Proactive Temp" or Proactive CPU Load".

Implementation

To implement the threshold, first we have to add the threshold property to the model section, in this case we name it “**env_temp**”, in the next step we will be using this name to link the model to the threshold attributes and to the stats attribures.

```
'systemHealth' => {
  --snip—
  'rrd' => {
    'env_temp' => {
      'indexed' => 'true',
      'threshold' => 'env_temp',
      --snip—
    }
  }
}
```

We add the threshold values to Common-threshold.nmis, using the name specified before. The event name must include "Proactive" at the beginning

```
'env_temp' => {
  'item' => 'currentTemp',
  'event' => 'Proactive Temp',
  'select' => {
    'default' => {
      'value' => {
        'fatal' => '90',
        'critical' => '80',
        'major' => '70',
        'minor' => '60',
        'warning' => '50'
      }
    }
  }
},
```

Next we add statistics extraction to Common-stats.nmis

```
'env_temp' => [
  'DEF:currentTemp=$database:currentTemp:AVERAGE',
  'PRINT:currentTemp:AVERAGE:currentTemp=%1.2lf',
```

Once we have created the threshold, it is time to tested. The best way to test if it is working as desired, is by using nmis.pl.

```
$ nmis.pl type=thresholds debug=true
```

To trigger the threshold, we need to modify the current threshold to be under current temperature.

```
'env_temp' => {
  'item' => 'currentTemp',
  'event' => 'Proactive Temp',
  'select' => {
    'default' => {
      'value' => {
        'fatal' => '90',
        'critical' => '80',
        'major' => '70',
        'minor' => '60',
        'warning' => '5'
      }
    }
  }
},
```

To test it we run nmis.pl type=thresholds, and we verify that the events has been created. After that, we return the value to it's previous state, running once again nmis.pl type=thresholds, the event should be closed now.

Standard Thresholds (Common)

NMIS includes a set of standard thresholds which are commonly associated to some vendors. This is a summary of these thresholds.

Treshold Name	Event	Vendor
available	Proactive Interface Availability	Common for all Vendors
calls_util	Proactive Calls Utilisation	Cisco
ccpu	Proactive CPU	Cisco
cpu	Proactive CPU	Cisco (the most common for Cisco devices)
cpuUtil	Proactive CPU	Alcatel, Zyxel
cpu_cpm	Proactive CPU	Cisco
env_temp	Proactive Temp	Cisco, Zyxel
hrsmpcpu	Proactive CPU	Microsoft
jnx_buffer	Proactive Buffer Utilisation	Juniper
jnx_cpu	Proactive CPU	Juniper
jnx_heap	Proactive Heap Utilisation	Juniper
jnx_temp	Proactive Temp	Juniper
mem-proc	Proactive Memory Free	Cisco
memUtil	Proactive Memory Utilisation	Alcatel, Zyxel
modem_dead	Proactive Dead Modem	Cisco
modem_unav	Proactive Modem Utilisation	Cisco
pkt_discards_in	Proactive Interface Discards Input Packets	Common for all Vendors
pkt_discards_out	Proactive Interface Discards Output Packets	Common for all Vendors
pkt_errors_in	Proactive Interface Error Input Packets	Common for all Vendors
pkt_errors_out	Proactive Interface Error Output Packets	Common for all Vendors
reachable	Proactive Reachability	Common for all Vendors
response	Proactive Response Time	Common for all Vendors
ssCpuRawIdle	Proactive CPU IO Idle	net-snmp (Linux, Solaris, etc)
ssCpuRawSystem	Proactive CPU IO System	net-snmp (Linux, Solaris, etc)

ssCpuRawUser	Proactive CPU IO User	net-snmp (Linux, Solaris, etc)
ssCpuRawWait	Proactive CPU IO Wait	net-snmp (Linux, Solaris, etc)
util_in	Proactive Interface Input Utilisation	Common for all Vendors
util_out	Proactive Interface Output Utilisation	Common for all Vendors

Creating Thresholds (Detailed)

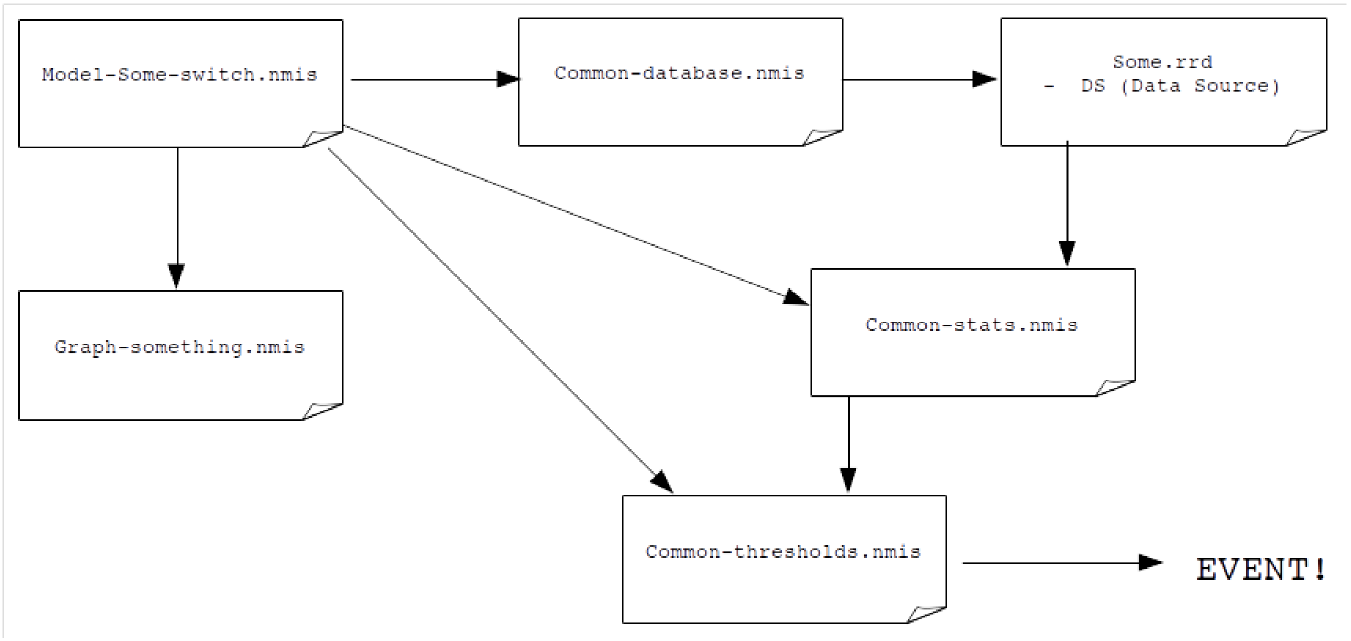
Files

Files that require modification:

- /usr/local/nmis8/models/Model-Some-switch.nmis
- /usr/local/nmis8/models/Common-database.nmis
- /usr/local/nmis8/models/Common-header.nmis
- /usr/local/nmis8/models/Common-stats.nmis
- /usr/local/mmis8/models/Common-threshold.nmis

Relationship

Relationship of the files to each other, it may be useful to visualise how the files interact with each other.



Common Attributes

There are several common attributes that must match between these files in order for thresholding to work. In an attempt to demonstrate the relationship between these variables we'll use the following labels. Please reference the code block below for where they should reside.

alpha - This is the threshold variable set in the Model-Some-switch.nmis file and must be referenced in the name section of the Common-threshold nmis file.

bravo - This the the graphtype variable set in the Model-Some-switch.nmis file. This sets the name of the graph that NMIS will invoke for a particular RRD file. In order for the graph heading to be correct this variable with its associated description needs to be added to the Common-heading.nmis file.

charlie - This is the name of the RRD file that the data retrieved via SNMP will be written to. This variable is set in the Model-Some-switch.nmis file. It must also be referenced in the Common-database.nmis file in order for the RRD to be created and updated in the correct directory. This variable is referenced in the Common-stats.nmis file in order for NMIS to know where to find statistical data.

delta - This is specific to an SNMP OID and is set in the Model-Some-switch.nmis file. This will be the 'DS' (Data Source) title within the RRD file that will contain the actual data returned for a specific OID. This variable is referenced in the Common-stats.nmis file in order to provide the DS for the specific data that needs to be presented.

echo - This variable is set in the Common-stats.nmis file. It is used to make computations on the zebra variable in the RRD language. This variable is then passed to the Common-threshold.nmis file in order to fire and event.

```
### Model-Some-Switch.nmis

%hash = (

     'systemHealth' => {

          'rrd' => {

               '<charlie>' => {

                    'graphtype' => '<bravo>',

                    'indexed' => 'true',

                    'threshold' => '<alpha>'

                    'snmp' => {

                         '<delta>' => {

                              'oid' => 'hrProcessorLoad',

                              'option' => 'gauge,0:U'

                         }

                    }

               }

          }

     }

)
```

```
### Common-threshold.nmis

%hash = (

     'threshold' => {

          'name' => {

               '<alpha>' => {

                    'item' => '<echo>',

                    'event' => 'Proactive CPU',

                    'select' => {

                         'default' => {

                              'value' => {

                                   'fatal' => '90'

                                   'critical' => '80',

                                   'major' => '70',

                                   'minor' => '60',

                                   'warning' => '50'

                              }

                         }

                    }

               }

          }

     }

)
```

```
### Common-database.nmis

'<charlie>'   =>   '/nodes/$node/health/<charlie>-$index.rrd',
```

```
### Common-stats.nmis

%hash = (

     'stats' => {

          'type' => {

               '<charlie>' => {

                    'DEF:<echo>=$database:<delta>:AVERAGE',

                    'PRINT:<echo>:AVERAGE:<echo>=%1.0f',

               }

          }

     }

)
```

```
### Common-heading.nmis

'<bravo>' => 'Processor Load',
```

Thresholds Controls

Simple Thresholds

In NMIS a simple threshold is defined by the following:

- the name
- the event name (which must begin with the phrase "Proactive" for correct event handling)

- a select (with a default and optionally more)
- default threshold values

In the file /usr/local/nmis8/models/Common-threshold.nmis this looks like this:

```
'cpu' => {
  'item' => 'avgBusy5min',
  'event' => 'Proactive CPU',
  'select' => {
    'default' => {
      'value' => {
        'critical' => '70',
        'fatal' => '80',
        'minor' => '50',
        'warning' => '40',
        'major' => '60'
      }
    }
  }
},
```

Have a set of thresholds for Core CPU

However, Core devices are more sensitive to CPU Load. So we want to use a different set of threshold values. Something like:

> 'critical' => '60',
>
> 'fatal' => '70',
>
> 'minor' => '40',
>
> 'warning' => '30',
>
> 'major' => '50'

But how to make these apply just to Core devices?

Advanced Thresholds with Controls

For example, different thresholds for core devices. Looking in Common-thresholds will give you some ideas, but you can add many “selects” and have properties like:

- $name
- $node
- $host
- $group
- $roleType
- $nodeModel
- $nodeType
- $nodeVendor
- $sysDescr
- $sysObjectName
- others for interface
- Almost unlimited possibilities.

So we can create a more specific threshold for core devices (NMIS has this already configured by default).

```
'cpu' => {
  'item' => 'avgBusy5min',
  'event' => 'Proactive CPU',
  'select' => {
    '10' => {
      'value' => {
        'critical' => '60',
        'fatal' => '70',
        'minor' => '40',
        'warning' => '30',
        'major' => '50'
      },
      'control' => '$roleType =~ /core/'
    },
    --snip--
    'default' => {
      'value' => {
        'critical' => '70',
        'fatal' => '80',
        'minor' => '50',
        'warning' => '40',
        'major' => '60'
      }
    }
  }
},
```

These are executed in the select order, and if no control is matched, then the default set is used.

Advanced Control Options

The following are the available control options:

Node Properties

- $name
- $node
- $host
- $group
- $roleType
- $nodeModel
- $nodeType
- $nodeVendor
- $sysDescr
- $sysObjectName

Indexed Objects like interfaces

- $ifAlias
- $Description
- $ifDescr
- $ifType
- $ifSpeed
- $ifMaxOctets
- $maxBytes
- $maxPackets
- $entPhysicalDescr

Newly added indexed objects in NMIS 8.6G

- $hrStorageDescr
- $hrStorageType
- $hrStorageUnits (disk block size)
- $hrStorageSize (disk size in blocks)
- $hrStorageUsed (disk used in blocks)
- $hrDiskSize (disk size in bytes, hrStorageSize * hrStorageUnits)
- $hrDiskUsed (disk used in bytes, hrStorageUsed * hrStorageUnits)
- $hrDiskFree (disk free in bytes)

Sample Controls

The controls are little pieces of code which will be evaluated when needed, so you might want to do the following sorts of things.

Result	Control
Use this threshold if the interface speed is between 1 and 5 megabits/second	$ifSpeed <= 50000 and $ifSpeed >= 10000
Use this threshold if the interface speed is 100 megabits	$ifSpeed == 100000000
Use this threshold if the interface speed is 10 megabits	$ifSpeed == 10000000
Use this threshold if the interface speed is 1 gigabits	$ifSpeed == 1000000000
Use this threshold if the disk is larger than 100 gigabytes	$hrDiskSize >= 104857600000
Apply the threshold to all devices starting with the IP address 192.168	$host =~ /192\.168/
Apply the threshold to all devices in the group "Sales"	$group eq "Sales"
Apply the threshold to all Cisco IOS devices	$sysDescr =~ /Cisco IOS/

Alerting

Basic Alerts

An alert is a custom event generated by testing the value of an OID or custom variable and producing a boolean result (true or false). If the test returns true, an event is raised and it will run through the escalation system, false will not raise an alert. Later on, when the test that was returning true once again returns false the event will be cleared.

The values required for the alert are:

test => this is a boolean operation using $r to determine if the alert should be raised
event => the name of the event when it is raised
level => the level of the event when it is raised

Test

This can be any Perl expression, and its evaluation result will be interpreted like perl does booleans (i.e. empty string, 0, undef means false, anything else means true).

$r holds the value of the oid and you need to use $r to test for the condition you want to raise the alert for. All of perl's operators are available, the most likely suspects are:

```
# Numbers
"==" returns true if the left argument is numerically equal to the right argument.
"!=" returns true if the left argument is numerically not equal to the right argument.
"<" returns true if the left argument is numerically less than the right argument.
">" returns true if the left argument is numerically greater than the right argument.
"<=" returns true if the left argument is numerically less than or equal to the right.
">=" returns true if the left argument is numerically greater than or equal to the right.

#strings
"eq" returns true if the left argument is stringwise equal to the right argument.
"ne" returns true if the left argument is stringwise not equal to the right argument.
"lt" returns true if the left argument is stringwise less than the right argument.
"gt" returns true if the left argument is stringwise greater than the right argument.
"le" returns true if the left argument is stringwise less than or equal to the right argument.
"ge" returns true if the left argument is stringwise greater than or equal to the right argument.
```

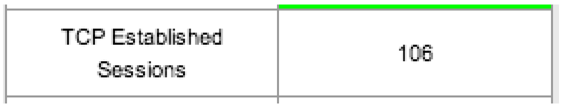
A quick note on stringwise versy numerical comparison: in numeric mode, the expressions will be converted to numbers (i.e. string "0003" becomes the number 3 for comparison). In string mode the expressions' characters are compared one by one. If $r is "0003", $r == 3 is true, but $r eq 3 is false.

Where to add the Alert

The alert can be added to current variables being polled from the devices, or a new section can be added. For example a new section in Model->system->sys could be added which might look like the example below (the "--snip--" indicates that extra model code has been removed for clarity):

```
%hash = (
  --snip--
  'system' => {
     --snip--
    'sys' => {
      'standard' => {
        --snip--
      },
      'alerts' => {
        'snmp' => {
          'tcpCurrEstab' => {
            'oid' => 'tcpCurrEstab',
            'title' => 'TCP Established Sessions',
            'alert' => {
              'test' => '$r > 250',
              'event' => 'High TCP Connection Count',
              'level' => 'Warning'
            }
          }
        }
      },
       --snip--
    }
    --snip--
  }
  --snip--
);
```

Adding the alert also adds the information to the "Device Details" panel, so you get the last polled value displayed all the time. Note that when you add such a basic alert its variable is collected independently of any other variables that your model might collect.



TCP Established Sessions	106

Example

The following is an example of the layout of an alert (in this example serialNum is taken from Model-CiscoRouter.nmis) and uses a string based (stringwise) comparison:

```
'serialNum' => {
  'oid' => 'chassisId',
  'title' => 'Serial Number',
  'alert' => {
    'test' => '$r ne "SomeSerialNumber"',
    'event' => 'Serial Number Invalid',
    'level' => 'Critical'
  }
}
```

$r in this case is the value of the OID chassisId. This will raise the event "ALERT: Serial Number Invalid" when the oid chassisId is not equal to "SomeSerialNumber".

A numeric based comparison can also be used, in this case when the value is 0 the alert is raised, when the value is not 0 the alert is cleared:

```
'cipSecGlobalActiveTunnels' => {
  'oid' => 'cipSecGlobalActiveTunnels',
  'title' => 'Global Active Tunnels',
  'alert' => {
    'test' => '$r == 0',
    'event' => 'No tunnels present',
    'level' => 'Critical'
  }
}
```

More Advanced Alerts

Alerts can also be created in the 'alerts' section of the model. lets created in this section have the advantage of being able to use values from a whole section of data to determine if the alert should be triggered or not; however, such alerts can NOT access variables collected/modelled in the 'system' section and as such are mostly useful for systemHealth modelling.

If the model does not have such an 'alerts' section, it can be added at the outermost level of the model, e.g. along side '-common=' and 'system'.

A concrete example always makes things more clear.

```
%hash = (
 '-common-' => {
   -- snip --
 },
 'system' => {
   -- snip --
 },
 'storage' => {
   -- snip --
 },
 'alerts' => {
   'services' => {
     'HighProcessMemoryUsage' => {
       'type' => 'test',
       'test' => 'CVAR1=hrSWRunPerfMem;$CVAR1 > 300000',
       'value' => 'CVAR1=hrSWRunPerfMem;$CVAR1 * 1',
       'unit' => 'KBytes',
       'element' => 'hrSWRunName',
       'event' => 'High Process Memory Usage',
       'level' => 'Warning'
     }
   }
 }
);
```

Let's break down the above example.

- 'services' defines what section the values being used for the alert are taken from. In this case services won't be found in the model because it is a special section just for servers. Normally you will not need to worry about special sections. Please note that you CANNOT use the 'system' section for advanced alerts!
- 'HighProcessMemoryUsage': this creates a label/id for the alert
- 'type' => 'test': this means the alert will test a single condition. The options are ['test', 'threshold-rising', 'threshold-falling']
- 'test' => 'CVAR1=hrSWRunPerfMem;$CVAR1 > 300000' defines a custom variable and then uses that variable to perform a boolean test.
- See the paragraph below regarding custom variables.
- 'value' => 'CVAR1=hrSWRunPerfMem;$CVAR1 * 1': this defines how the value that triggered the alert should be reported and displayed when the alert is shown in the GUI
- 'unit' => 'KBytes': the unit that the above value will be displayed with
- 'element' => 'hrSWRunName': which OID/value that has the problem, a descriptor or identifier. In this case it is showing the name of the process that has high memory usage.
- 'event' => 'High Process Memory Usage': sets the name of the alert event
- 'level' => 'Warning': the level the event will be triggered with. When using thresholding this is not used as the thresholds define the level.

Custom Variables

Please note that in NMIS versions before 8.6 you can only use one custom variable in a test expression, namely CVAR. This limitation has been removed in NMIS 8.6, and the limitation also never applied to value or control expressions.

As described in more detail on the Advanced Modelling page, your test, control and value expressions may contain custom variable definitions and accesses. The syntax is straightforward:

`CVAR=snmp_var_name;` or `CVAR3=other_snmp_var;` looks for the named snmp object value (which must be listed for collection in the model!) and saves it in one of eleven custom variables (CVAR and CVAR0 to CVAR9). When you want to access that saved value, you use `$CVAR` , `$CVAR3` etc.

Please note that the substitution is performed on a purely textual basis before perl is given the expression to evaluate; if you want to use string variable contents you should double-quote your access expressions, e.g.

```
CVAR2=ifAlias; "$CVAR2" =~ /some description/
```

Thresholding VS Alerting

When to use

Running a Model

Debugging Device Modelling

Advanced Modelling

Advanced Modelling Options

Regex OID

Control

Calculate

ParseString

Replace

The result of the collection can be replaced by a given value from a predefined lookup table. In this case the value 1 or 0 will be replaced by the string "yes" or "no".

e.g: `replace => {`

```
      '1' => 'yes',
      '0' => 'no'
    }
```

No graphs

If we do not have the need to display a graph as part of a table of the systemHealth section, the option 'graphtype' must be replaced by: `'no_graphs' => '1'`

Not saving to RRD (nosave)

If data is collect using snmp by the model to be displayed but there is no need to save it to RRD, the option "nosave" should be used.

e.g: `'option' => 'nosave'`

Please note that setting `nosave` disables alerts for the given object.

index_regex

Used in the "SystemHealth" section, allows multi-element indexing: normally SNMP tables are indexed by the last, single numeric OID component. When NMIS does an update on a indexed entity, it iterates through all the known values for this index component and records them. This iteration does not work if the index consists of more than one number, as it does on certain equipment. In such cases you can set `index_regex` to a value that captures the OID components that vary between table elements. For example,

```
'index_regex' => '\.(\d+\.\d+\.\d+)$',
```

ensures that the last three numbers are used for indexing.

Additional Options

calculate

Format: expression

The result of the evaluated expression replaces the originally collected value.

control

Format: expression

The control expression will be evaluated when the respective section is consulted.

event

Format: string

Name of event.

format

Format: string

Printf format string for rewriting the collected value.

graphtype

Format: comma separated list

List of graph names. For each graph type there must be a graph definition file models/Graph-<graphtype>.nmis

indexed

Format: SNMP/WMI variable name (or true)

Declares that this subsection is indexed by the value of the given variable.

level

Format: severity level

Fatal, Critical, Major, Minor, Warning or Normal (in descending order)

logging

Format: boolean

Whether an event should be logged or not.

oid

Format: SNMP oid

Must contain a known name or a numeric OID.

option

Format: string

Declares the data type for this RRD Data Source, or prevents the variable from being saved if set to "nosave". If not defined, the default is 'gauge,U:U'.

order

Format: number

Order of processing, which always proceeds from lowest to highest number.

replace

Format: lookup table of known values and their replacement

The result of the collection can be replaced by a given value from this lookup table. If the value is not known, "unknown" is tried. If no replacement can be found, then the original value is left.

statstype

Format: string

Name for this type in the stats section.

stsname

Format: string

Name of parameter (...:stsname=...) in rrd rules in stats section.

sumname

Format: string

Name of parameter in summary file.

threshold

Format: comma separated list of names

Threshold names must be declared in the stats section.

title

Format: string

Used as label for displaying this variable.

value

Format: expression

The result of evaluated expression replaces the originally collected value.

Advanced Modelling: Custom variables

Occasionally you will come across a device or a situation where collecting a single SNMP variable is insufficient, for example when two or more SNMP properties need to be combined to provide a meaningful measurement.

NMIS version 8.4.8G and later support modelling such scenarios using custom variables, or `CVAR`s. With this mechanism you can temporarily capture up to 10 separate SNMP properties as a `CVAR` and define an arbitrarily complex expression (in perl) that transforms these `CVAR`s into the one measurement that you want to collect and/or display.

Where and How to use CVARs

`CVAR`s are supported

- in the `test` and `value` expressions in the NMIS alert and threshold subsystem,
- in `calculate` expressions in the general modelling subsystem,
- and from NMIS version 8.6 on, also in `control` expressions evereywhere (in versions before that only a single `CVAR` was supported in `control`)

To use `CVAR`s you define the required `CVAR`s as holding a previously specified SNMP variable at the beginning of one of the supported expressions; Subsequently you can then reference the `CVAR` value in the part of the expression that calculates the desired value to be used by NMIS.

An example scenario

The DS3 MIB defines a variety or error counters for DS3 circuits like "dsx3CurrentLCVs" which are based on a 15 minute observation interval and reset automatically at the end of the interval. As the interval start and end is arbitrary and up to the device to set, just capturing the error counters themselves is not quite workable. However, the DS3 MIB also specifies the variable "dsx3TimeElapsed" that holds the seconds elapsed since the start of the current observation interval. Dividing the raw error counter by the number of seconds into the interval results in a normalised errors-per-second rate which works well for collection and display.

Here is an excerpt of the relevant model file:

```
'systemHealth' =>
{
    'sections' => 'ds3Errors',
    'sys' =>
    {
        'ds3Errors' =>
        {
            'indexed' => 'dsx3CurrentIndex',
            'index_oid' => '1.3.6.1.2.1.10.30.6.1.1',
            'headers' => 'ds3intf,ds3linestatus',
            'snmp' => {
                'ds3intf' => {
                    'oid' => '1.3.6.1.2.1.2.2.1.2', # ifDescr
                    'title' => 'DS3 Interface',
                },
                'ds3linestatus' => {
                    'oid' => '1.3.6.1.2.1.10.30.5.1.10', # dsx3LineStatus
                    'title' => "DS3 Line Status",
                    'calculate' => 'my @x; my %triggers=(1,"No Alarm",2,"Rx Remote Alarm",4,"Tx Remote Alarm",
8,"Rx AIS",16,"Tx AIS",32,"Rx LOF",64,"Rx LOS",128,"Loopback",256,"Test Pattern",512,"Unknown",1024,"Near end
unavailable signal",2048,"Carrier Equip OOS"); while (my ($num,$txt)=each(%triggers)) { push (@x,$txt) if (int
($r) & int($num)); }; return join(", ",@x); ',
                },
...
                'ds3LCV' => {
                    'oid' => '1.3.6.1.2.1.10.30.6.1.6', # dsx3CurrentLCVs
                    'title' => 'Line Coding Violations per second',
                    'calculate' => 'CVAR1=ds3Elapsed; return ($CVAR1? $r/$CVAR1 : 0);',
        },
    },  # sys

    'rrd' => {
        'ds3Errors' => {
            'indexed' => 'true',
            "graphtype" => "ds3Errors",
            "snmp" => {
                'ds3Elapsed' => {
                    'oid' => '1.3.6.1.2.1.10.30.5.1.3', # dsx3TimeElapsed
                    'title' => 'elapsed seconds in current measurement interval',
                    'option' => 'gauge,0:U',
                },
...
                "ds3LCV" => {
                    'oid' => '1.3.6.1.2.1.10.30.6.1.6',
                    'option' => 'gauge,0:U',
                    'title' => "Line Coding Violations per second",
                    'calculate' => 'CVAR1=ds3Elapsed; return ($CVAR1? $r/$CVAR1 : 0);',
                },
    },  # rrd
},  # systemhealth
```

In the example above, the `calculate` expressions are used in two ways:

- to transform the bitfield variable "DS3 Line Status" into a more verbose textual list of component statuses,
- and to divide the raw `dsx3CurrentLCVs` error count by the `dsx3TimeElapsed` interval length.

In both cases the syntax is very straight-forward:

- The expression must be a valid perl statement and return exactly one value.
- The tokens `$r`, and `CVAR0` to `CVAR9` are interpreted by NMIS; everything else is perl.
- Defining and using local variables with `my` is ok, but don't attempt to change any global NMIS variables.
- "`CVAR1=some_snmp_var;`" defines what SNMP object CVAR1 is supposed to hold. The parser understands `CVAR0` to `CVAR9` for a total of 10 captures.

- You can use functions that were defined elsewhere in NMIS in your `calculate` expression.
  You will likely have to include the full module namespace in the function call, e.g. `func::beautify_physaddress(...).`
  Only functions without side-effects should be used.
- "`return $r/$CVAR1;`" accesses the value of `CVAR1` in an expression. The variable "`$r`" represents the SNMP variable that the `calculate` expression is attached to.

Please note that

- the `$CVARn` replacement in the expression is performed on a purely textual basis, before the expression is handed to the perl interpreter for evaluation :
  - For string variables you have to provide quotes in your expression, e.g.

```
calculate => 'CVAR1=somestringthing; return 42 if ("$CVAR1" eq "online");'
```

- Numeric variables can be used straight without quotes.
- the `$CVARn` access refers to the *raw* value of the named property, ie. the data before any `replace` or `calculate` expressions for the named property were evaluated.

How to keep temporary CVAR data out of the RRD databases

As outlined above all the objects that you want to access via `CVAR`s must be defined in the same section. If your test/calculate expression is within an `rrd` section, all the other objects will have to be within that `rrd` section, too, and thus they would be collected by NMIS and stored in RRD - quite wasteful if these other variables are just temporary and only there to for access using one `CVAR` expression.

In versions 8.6.0 and above you can prevent this by adding an `option` with value `nosave`:

```
'snmp' => {
  'hrNumUsers' => {
     'oid' => 'hrSystemNumUsers',
     'option' => 'nosave',
  },
```

In the example above, `hrNumUsers` would be retrieved with SNMP, and other variables could be defined in terms of e.g. `CVAR3=hrNumUsers`, but `hrNumUsers` would not be saved.

Please note that setting `nosave` disables alerts for the given object

Plugins

NMIS8 Plugin Operation

NMIS8 Plugins will run for any node which is active, but logically a plugin needs to be able to know what type of node it is operating on. Plugins will generally include early in the code a statement to look for a specific model type and if the model is of a type interesting to it, it will perform its duties, otherwise it will skip (return nothing).

To have an API Only node, the best method is to set active to true, and ping, collect, collect_snmp and collect_wmi to false and manually change the model from "automatic" to be the name of the model which will be used by the plugin. Because there is no data provided by the device we do not know how to automatically discover how to talk to the device, NMIS needs a little nudge to know how to talk to the device. You can see an example of this with the CiscoMerakiCloud and CiscoViptelaCloud models and plugins, which work with Cisco Meraki and Cisco SDN WAN (Viptela) respectively.

Intro (Concepts)

Basic Plugins

Examples

******* Use CMD8TEMP plugin as example (https://support.opmantek.com/browse/SUPPORT-6061)

NMIS8 Node Polling Configuration

NMIS versions up to and including NMIS 8.6.7G

The following indicates how NMIS will behave when provided with various configuration options, this is specifically concerned with active, ping and collect.

Polling Description	active	ping	collect	plugins	services
Regular node polling, the node will be ICMP polled and will have SNMP and WMI (if credentials configured) data collected	true	true	true	Will be run	will be polled if configured

SNMP Only Node, the not will have SNMP collected but ICMP polling will not be performed.	true	false	true	Will be run	will be polled if configured
Ping Only Node, the node will only be polled using ICMP.	true	true	false	Will be run	will be polled if configured
Service Only Node, the node will only have services collected if they are configured.	true	false	false	Will be run	will be polled if configured
Node is NOT active and NMIS will mostly ignore the node.	false	N/A	N/A	N/A	N/A

NMIS versions after and including NMIS 8.6.8G

The following indicates how NMIS will behave when provided with various configuration options, this is specifically concerned with active, ping and collect.

Polling Description	active	ping	collect	collect_snmp	collect_wmi	services
Regular node polling, the node will be ICMP polled and will have SNMP and WMI (if credentials configured) data collected	true	true	true	true	true	will be polled if configured
SNMP or WMI Only Node, the not will have SNMP collected but ICMP polling will not be performed.	true	false	true	true	true	will be polled if configured
Ping Only Node, the node will only be polled using ICMP.	true	true	false	false	false	will be polled if configured
Service Only Node, the node will only have services collected if they are configured.	true	false	false	false	false	will be polled if configured
API Only Node, the node will use plugins to collect data, no other polling will be done, except services if configured.	true	false	false	false	false	will be polled if configured
Node is NOT active and NMIS will mostly ignore the node.	false	N/A	N/A	N/A	N/A	N/A

Model Policy

NMIS 8.6 introduced a new mechanism for adjusting a model's behaviour for particular nodes: the Model Policy system. In version 8.6.0G it allows you to specify flexible rules for adding or removing systemHealth model sections for specific nodes (or groups of nodes).

The Model Policy Document

The installer will install a default model policy document in conf/Model-Policy.nmis. The original/default file will also remain available in the install directory, and contains helpful comments.

The structure of the policy document is quite simple but fairly flexible:

- The policy consists of any number of rules.
- The rules are evaluated in order of their numeric key; fractional numbers are supported to simplify insertion.
- Only the first matching rule is applied.
- Each rule must express what changes to systemHealth should be made.
- The changes are expressed as a list of systemHealth section name plus the desired activation state (true or false).
- systemHealth sections not named are not modified and therefor default to True or on (so check the default section for what would otherwise be disabled).
- Each rule may include any number of filter expressions, which determine whether the rule should be applied to a particular node.
- All given filter expressions must match simultaneously for the rule to be considered a match.
- A filter expression defines a node property or configuration setting to be compared against an explicit list of acceptable values, or a regular expression.
- Node properties are given as node.<propname>, and for configuration settings you'd use conf.<configsetting>.
- All configuration settings are available, using the prefix conf. and the same names as seen in conf/Config.nmis.
- The available node properties are: the static ones from the node configuration, plus the more dynamic ones from the system section in the node's "node info" file (var/<nodename>-node.json).
- A Model Policy document may also include an extra section named _display, which controls in what order the default policy's entries should be shown in the Configuration GUI.
- See the default policy for an example.

It should be noted that as Only the first matching rule is applied and therefor the default rule is not subsequently applied you should include all the relevant "false" sections from the default rules into your rule. For example if you wanted to turn on just one mpls system health section you would set that as true in your rule and you would also include all the other "false" lines which are relevant to your model in the rule.

Example Policy

Here is a partial example policy:

```
%hash = (
         # rule numbers may be fractional numbers (for easy insertion)
         # first matching rule terminates the policy application
         10 => {
             # filter keys: node.xyz or config.abc; node.nodeModel is the (possibly dynamic) current model
             # filter values: string, list of strings,
             # or regexp (=string with //, optional case-insensitive //i)
             IF => { 'node.name' => ['node1','node2'],
                     'node.location' => '/def.*/',
                     'config.auth_ldap_server' => '/192\./', },
             # sections to adjust, only systemHealth supported so far
             systemHealth => {
                 'fanStatus' => 'true',    # add if not present
                 'tempStatus' => 'false', # remove if present
             },
         },
         20 => {
             IF => { 'node.name' => 'embedded' },
             systemHealth => {
                 diskIOTable => 'false' # this node runs off r/o flash disk
             },
         },
         999 => {    # the fallback/defaults, without filter
             systemHealth => {
                 cdp => 'true',
                 lldp => 'true',
                 bgpPeer => 'true',
                 ospfNbr => 'true', }
         } );
```

The first rule applies to at most two particular nodes (because of the given list of node.name values), and only if their location property starts with "def" and only if the NMIS configuration is set up for an LDAP server in the 192.0.0.0/8 network. For all systems that match these restrictions the fanStatus and tempStatus model sections are enabled.

The second rule disables the diskIOTable model section for a specific system that doesn't have real 'disks', just a readonly flash drive.

The last rule does not have any filtering IF clauses, therefore applies to all nodes and thus it serves to set the "default" policy. As mentioned above only the first matching rule is applied, hence the default rule will only apply to nodes where rules 10 and 20 have not matched.