

SNMP Tool Synagon

Synagon SNMP Graphing Tool

Overview

Synagon is Python-based program that collects and graphs firewall values from Juniper routers. The tool first requires the user to select a router from a list. It will retrieve all the available firewall filters/counters from that router and the user can select any of these to graph. It was developed by DANTE for internal use, but could be made available on request (on a case by case basis), in **an entirely unsupported manner**. Please contact operations@dante.org.uk for more details.

[The information below is taken from DANTE's internal documentation.]

Installation and Setup procedures

Synagon comes as a single compressed archive, that can be copied in a directory and then invoked as any other standard python script.

It requires the following python libraries to operate:

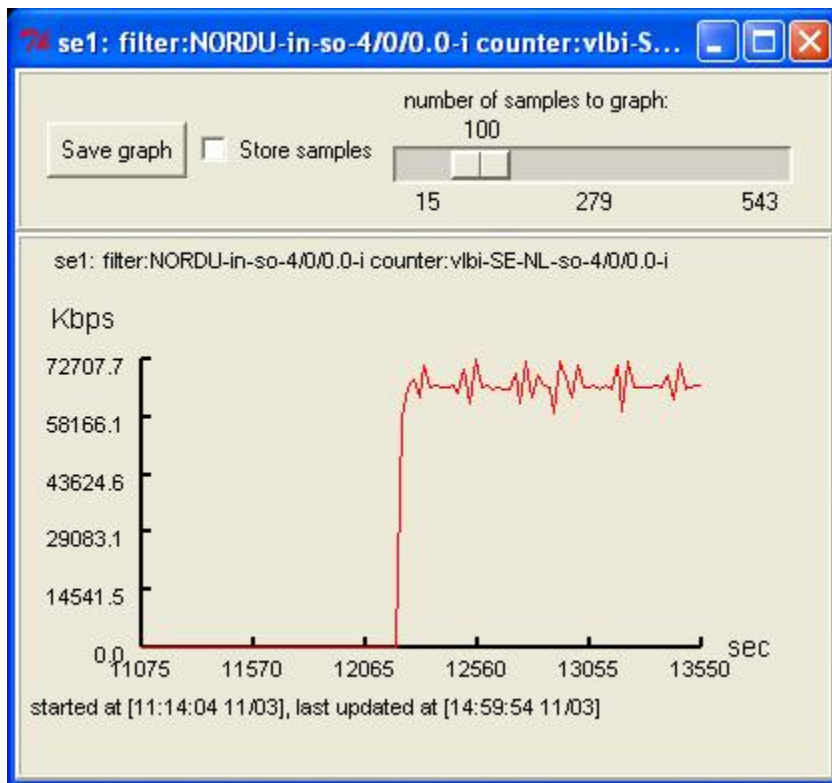
- 1) The python SNMP framework (available at <http://pysnmp.sourceforge.net/>)

The SNMP framework is required to SNMP to the routers.

For guidance how to install the packages please see the section below

You also need to create a network.py file that describes the network topology where the tool will act upon. The configuration files should be located in the same directory as the tool itself. Please look at the section below for more information on the format of the file.

- Example Sysngaon screen:



How to use Synagon

The tool can be invoked by double clicking on the file name (under windows) or by typing the following on the command shell:

```
python synagon.py
```

Prior to that, a `network.py` configuration file must have been created and located at the directory where the tool has been invoked. It describes the location of the routers and the `snmp` community to use when contacting them.

Immediately after, a window with a single menu entry will appear. By clicking on the menu, a list of all routers described in the `network.py` will be given as submenus.

Under each router, a submenu shows the list of firewall filters for that router (depending whether you have previously collected the list) together with a 'collect' submenu. The collect submenu initiates the collection of the firewall filters from the router and then updates the list of firewall filters in the submenu where the collect button is located.

The list of firewall filters of each router is also stored on the local disk and next time the tool is invoked it won't be necessary to interrogate the router for receiving the firewall list, but they will be taken instead from the file.

When the firewall filter list is retrieved, a submenu with all counter and policers for each filter will be displayed. A policer is marked with [p] and a counter with a [c] just after their name.

When a counter or policer of a firewall filter of a router is selected, the SNMP polling for that entity begins. After a few seconds, a window will appear that plots the requested values.

The graph shows the rate of bytes dropped since monitoring had started for a particular counter or the rate of packets dropped since monitoring had started for a particular policer.

The save graph button can create a file for the graph in encapsulated postscript format (.eps).

Because values are frequently retrieved, there are memory consumption and performance concerns. To avoid this a configurable limit is placed on the number of samples plotted on graph. The handle at the top of the window limits the number of samples plotted and has an impact on performance if is positioned at a high scale (of thousands).

Developer's guide

This section provides information about how the tool internally works.

The tool mainly consists of the felektis.py file, but it also relies on the external csnm.py and graph.py library. The felektis.py defines two threads of control; one initiates the application and is responsible for collecting the results, the other is responsible for the visual part of the application and the interaction with the user.

The GUI thread updates a shared structure (a dictionary) with contact information about all the firewalls the user has chosen by that time to monitor.

```
{{active_firewall_list }} {}=
```

The structure holds the following information:

router_name->filter_name->counter_name(collect, instance, type, bytes, epoch)

There is an entry per counter and that entry has information about whether the counter should be retrieved (collect), the SNMP instance (instance) for that counter, whether it is a policer or a counter (type), the last byte counter value (bytes), the time the last value was received (epoch).

It operates as follows:

```
do forever:
    if the user has shutdown the GUI thread:
        terminate

    for each router in active_firewall_list:
        for each filter of each router:
            for each counter/policer of each filter:
                retrieve counter value
                calculate change rate
                pass rate of that counter to the GUI thread
```

It uses SNMP to request the values from the router.

The gui thread (MainWindow class) first creates the user interfaces (based on the Tkinter Tcl/Tk module) and is passed the geant network topology found in the network.py file. It then creates a menu list with all routers found in the topology. Each router entry has a submenu with an item of 'collect'. This item is bound to a function and if it is selected, the router is interrogated via SNMP on the Juniper Firewall MIB, and the collection of filter name/counter names for that router is retrieved. Because the interrogation may take tens of seconds, the firewall filter list is serialised onto the local directory using the cPickle builtin module. The file is given the name router_name.filter (e.g.: uk1.filters for router uk1) and it is stored in the local directory where the tool was invoked from. Next time the tool is invoked, it will check if there is a file list for that router, and if so, it will deserialise the list of firewall filters from the file and populate the routers' submenu. It could of course be the case the serialised firewall list can be out of date, though the 'collect' submenu entry for each router is always there and can be invoked to replace the current list (both on memory and disk) with the latest details. If the user selects a firewall or policer to monitor from the submenu list, it will populate the active_firewall_list with that counter/policer, and the main thread will take it up and begin retrieving data for that counter/policer.

The main thread retrieves and passes the calculated rate to the MainWindow thread onto a queue that the main thread populates and the gui thread gets its data from. Because the updates can come from a variety of different firewall filters the queue is indexed with a text string which is a unique identifier based upon router name, firewall filter name, counter/policer name.

The gui thread periodically (i.e.: every 250ms) check if there are data in the above mentioned queue, If the identifier hasn't been seen before, a new window is created where the values are plotted. If the window already exists, then it is updated with that value and its contents are redrawn. A window is an instance of the class PlotList.

When a window is closed by the user, the gui modifies the active_firewall_list for that policer by setting the 'collect' attribute to None, and the main thread next time will reach that counter/policer, it will ignore it.

Creating a single synagon package

The tool consists of several files:

-

felegktis.py [main logic]

-

`graph.py` [graph library]

-

`csnmp.py` [SNMP library]

It is possible by using the squeeze tool (available at <http://www.pythonware.com/products/python/squeeze/index.html>) to make this file into a compressed python executable archive.

The `build_synagon.bat` uses the squeeze tool to archive all this file into one.

`build_synagon.bat`:

REM build the sinagon distribution

```
python squeezeTool.py -l -o synagon -b felegktis felegktis.py graph.py csnmp.py
```

The command above builds `synagon.py` from files `felegktis.py`, `graph.py` and `csnmp.py`.

Appendices

A. *network.py*

`network.py` describes the network's topology

- its format can be deduced by inspection. Currently the network topology name (graph name) should be set to 'geant' - the script could be adjusted to change this behaviour.

Compulsory router properties:

-

type: the type of the router [juniper, cisco, etc] etc (the tool only operates on 'juniper' routers)

- address: The address where the router can be contacted at
 - community: the SNMP community to authenticate on the router
- Example:

```

# Geant network
geant = graph.Graph()

# Defaults for router
class Router(graph.Vertex):
    """ Common options for geant routers """

    def (self):
        # Initialise base class
        graph.Vertex.(self)
        # Set the default properties for these routers
        self.property['type'] = 'juniper'
        self.property['community'] = 'compasspassword'

uk1 = Router()
gr1 = Router()

geant.vertex['uk1'] = uk1
geant.vertex['gr1'] = gr1

uk1.property['address'] = 'uk1.uk.geant.net'
gr1.property['address'] = 'gr1.gr.geant.net'

```

B. Python Package Installation

Python package installation

Python has a built in support for installing packages.

Packages usually come in a source format and they compiled at the time of the installation. Packages can be complete python source code or have extension in other languages (c, c++) . The packages can also come in binary form.

Many python packages which have extension written in some other language, need to be compiled into a binary package before distributing to platform like windows, because not all windows machine have c or c++ compilers that a package may require.

The Python SNMP framework library

The source version can be downloaded from <http://mysnmp.sourceforge.net>

When the file is uncompressed, the user should invoke the setup.py:

setup.py install

There is also a binary version for Windows that exists in the NEP's wiki. It has a simple GUI. Just keep pushing the 'next' button until the package is installed.

– Main.TobyRodwell - 23 Jan 2006