



# Zenoss Discovery and Classification

*April 2009*

*Jane Curry*

*Skills 1st Ltd*

[www.skills-1st.co.uk](http://www.skills-1st.co.uk)

Jane Curry  
Skills 1st Ltd  
2 Cedar Chase  
Taplow  
Maidenhead  
SL6 0EU  
01628 782565

[jane.curry@skills-1st.co.uk](mailto:jane.curry@skills-1st.co.uk)

# Synopsis

Zenoss has several possibilities for discovering devices, both manual and automatic. Once discovered, the subsequent monitoring of a device depends very much on the **device class** that an element is allocated to. This paper focuses particularly on a scenario that automatically discovers devices in networks and then allocates those devices to device classes.

The scenario uses a number of Zenoss techniques including Python scripts, event commands and event transforms.

## Table of Contents

1	Introduction.....	4
2	Automatic device class allocation scenario.....	9
3	Elements of the solution.....	12
3.1	dev_to_class.py scheduled script for device class allocation.....	12
3.2	Devices that do not (initially) support SNMP.....	14
3.3	SNMP agent installed subsequently for device in /Ping.....	18
4	Conclusions.....	22

# 1 Introduction

Zenoss provides a number of methods for discovering devices and their components. The simplest method is to manually add individual systems but this technique obviously does not scale well. If manual discovery is used, then many characteristics of the device can also be specified.

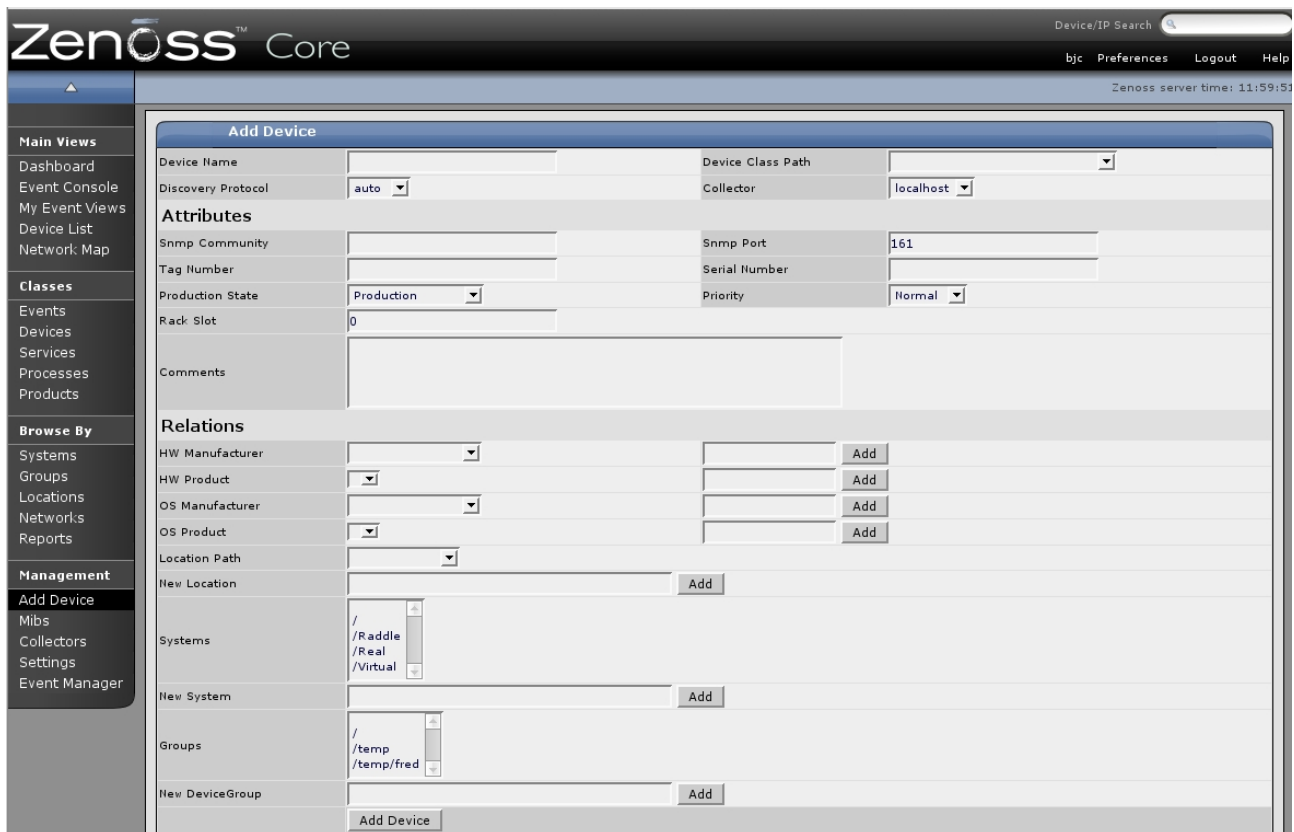


Figure 1: Dialogue for manually adding a new device

It is also possible to discover all devices on a network – this uses a ping-spray mechanism that adds devices which respond, so can be good if networks are subnetted to Class C with upto 254 devices, but it is not good if there are class B networks! Both techniques can either be driven from the Zenoss Graphical User Interface (GUI) or can be driven from the command line using *zendisc*. If devices are discovered by running discovery on a network, they are automatically added to the device class of */Discovered*.

Device classes generally control the availability and performance monitoring of a device. All device classes have **zProperties** that control SNMP access, telnet and ssh access, Windows Management Instrumentation (WMI) access and most other properties necessary for configuring monitoring.

zProperties Configuration				
Property	Value	Type	Path	
zCollectorClientTimeout	180	int	/	
zCollectorDecoding	latin-1	string	/	
zCollectorLogChanges	True	boolean	/	
zCollectorPlugins	Edit	lines	/	
zCommandCommandTimeout	15.0	float	/	
zCommandCycleTime	60	int	/	
zCommandExistenceTest	test -f %s	string	/	
zCommandLoginTimeout	10.0	float	/	
zCommandLoginTries	1	int	/	
zCommandPassword		string	/	
zCommandPath	/opt/zenoss/libexec	string	/	
zCommandPort	22	int	/	
zCommandProtocol	ssh	string	/	
zCommandSearchPath		lines	/	
zCommandUsername		string	/	
zDeviceTemplates	Device	lines	/	
zFileSystemMapIgnoreNames		string	/	
zFileSystemMapIgnoreTypes		lines	/	
zIcon	/zport/dmd/img/icons/noicon.png	string	/	
zIfDescription	False	boolean	/	
zInterfaceMapIgnoreNames		string	/	
zInterfaceMapIgnoreTypes		string	/	
zIpServiceMapMaxPort	1024	int	/	
zKeyPath	~/ssh/id_dsa	string	/	
zLinks		string	/	
zLocalInterfaceNames	^lo	string	/	

Figure 2: zProperties for the top-level /Device class (partial panel)

Note the *zCollectorPlugins* property (3 from the top) can be used to control the information that will be collected from a device on a **modelling** cycle (as opposed to a discovery poll).

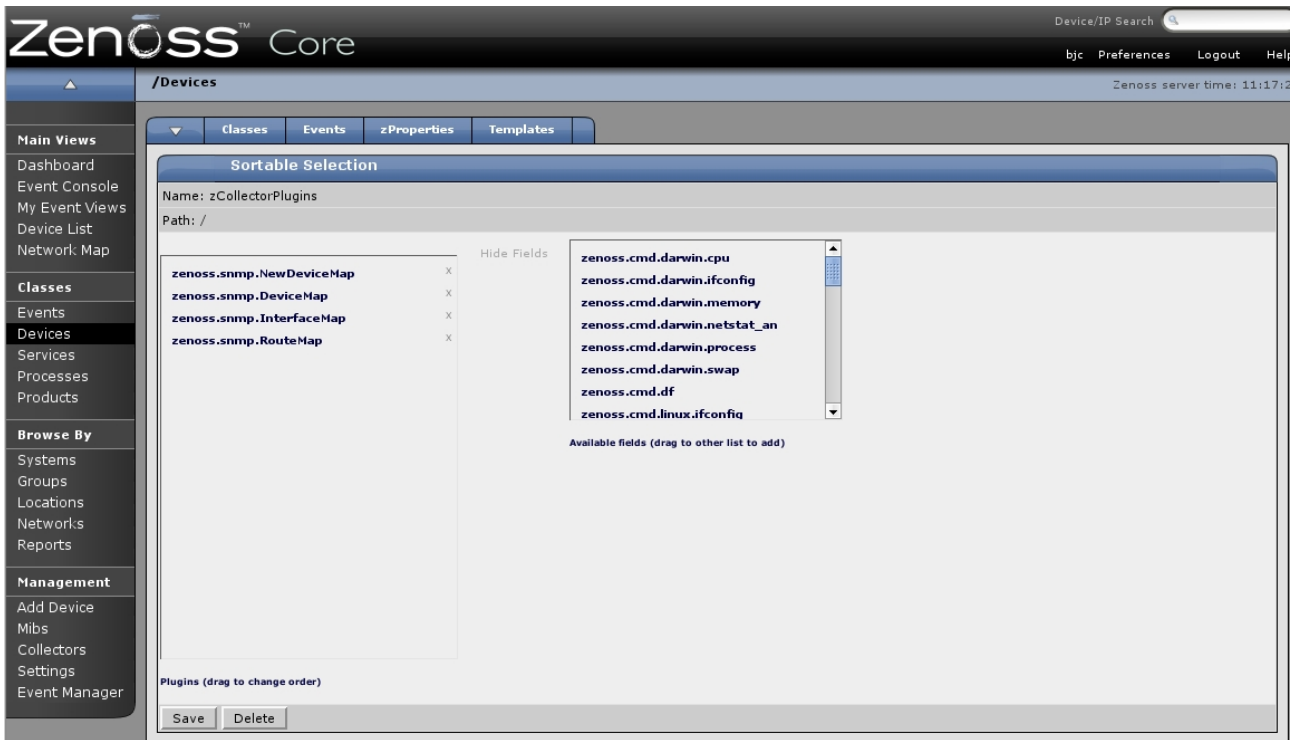


Figure 3: zCollectorPlugins for /Devices device class

The third element affected by device class is the performance Template. This controls the performance data that is collected and any thresholds that may generate events, based on that performance data.

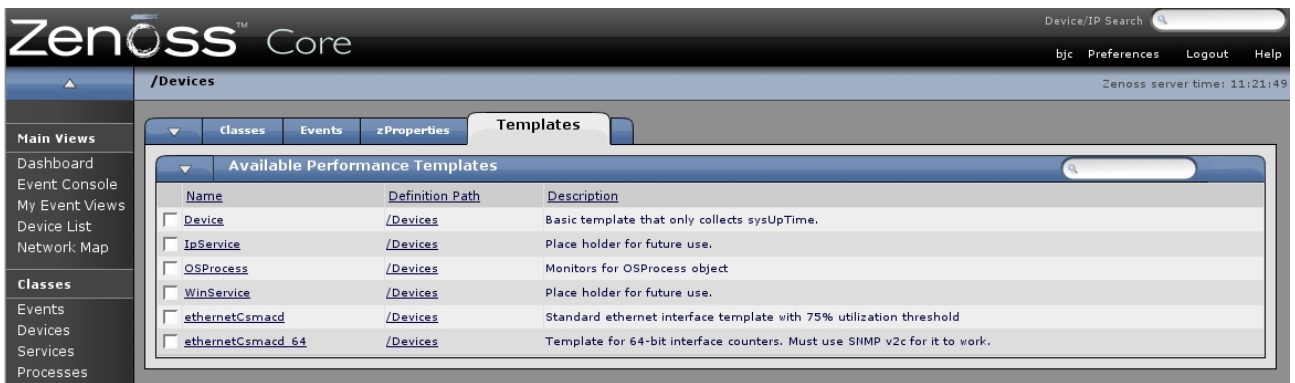


Figure 4: Performance Templates for /Devices

All device classes and individual devices have a zProperties page that can override any default or inherited zProperty. Zenoss's object-oriented class hierarchy means that common properties can be specified higher up the class hierarchy with specific attributes being overridden further down the hierarchy. So, for example, the device class /Server/Linux has extra zCollectorPlugins defined.

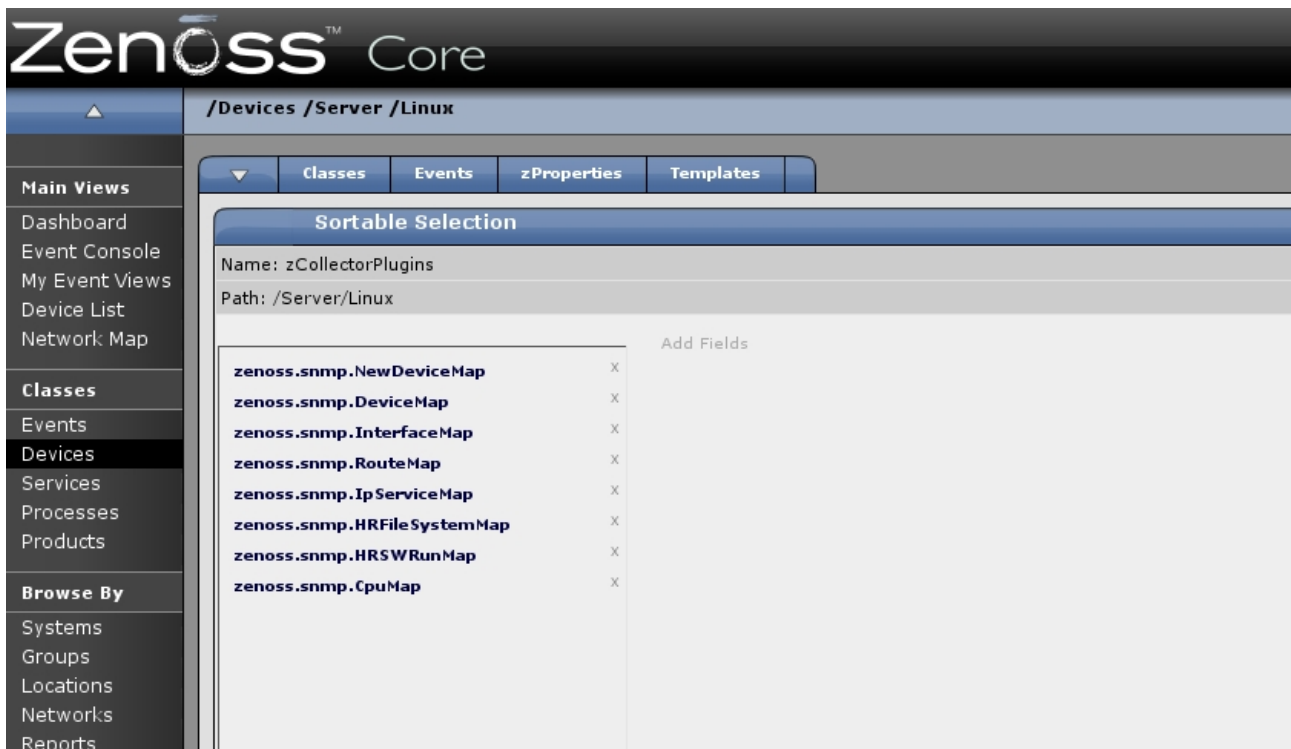


Figure 5: zCollector plugins for /Server/Linux

Any device that is placed in this class or a subclass of `/Server/Linux` will, by default, have CPU utilisation collected along with filesystem and software information from the SNMP Host Resources MIB. Note that the `zCollectorPlugins` are **not** specifying performance data; they are specifying availability information.

The standard performance template (called **Device**) for `/Devices`, is also overridden at the `/Server/Linux` subclass.

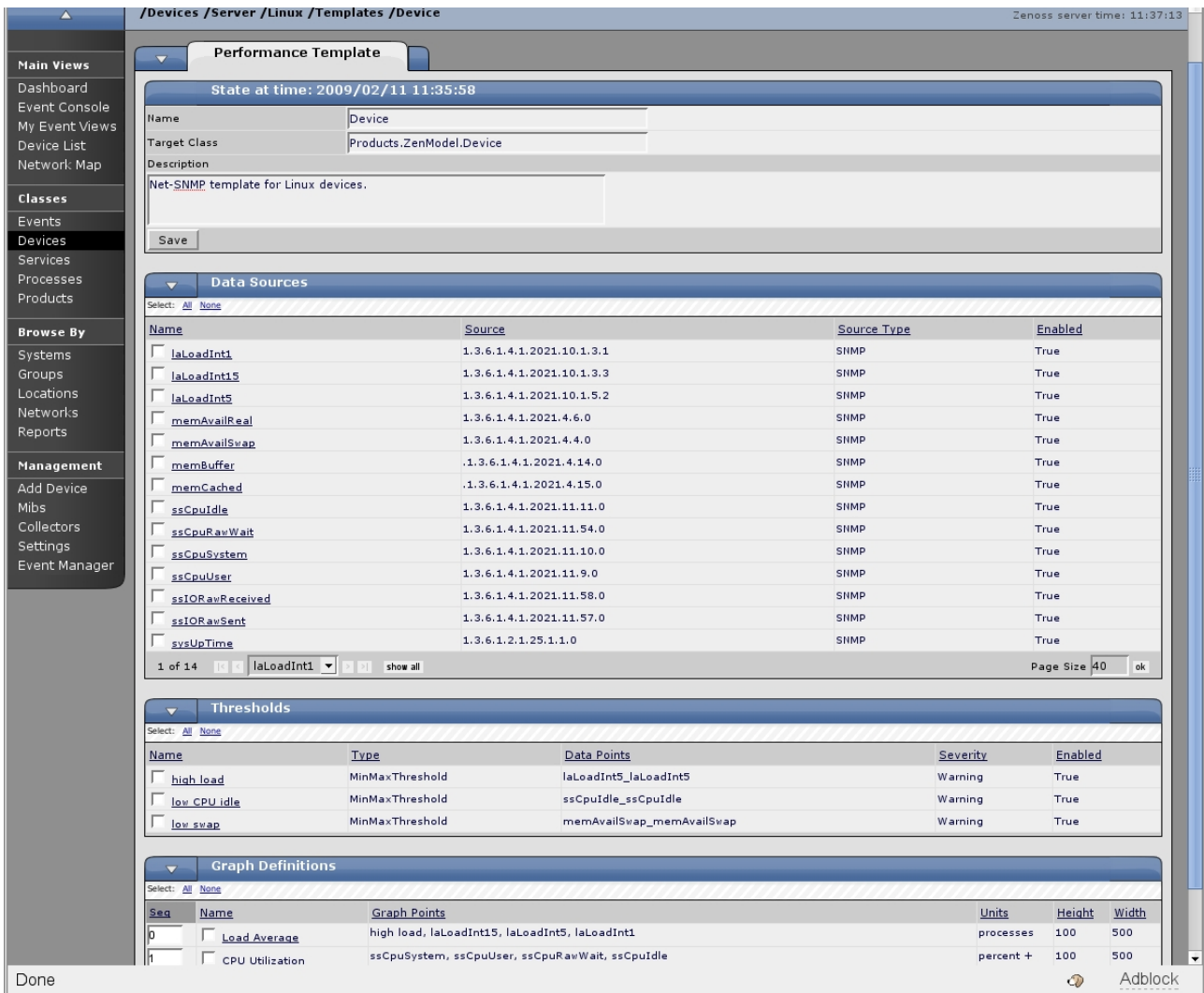


Figure 6: Performance template (called Device) for the /Server/Linux device class

Devices that are placed into /Server/Linux automatically have CPU, memory and IO data collected, graphs defined, and thresholds set so that events are generated for extremes of CPU utilisation and low swap.

Any of these availability or performance monitoring characteristics can be overridden either by a device subclass or by a specific device instance. Note that most default information collection relies on SNMP support.

The /Discovered device class, to which automatically discovered devices are added, has the same characteristics as the top-level /Devices class:

- ping polling is active
- SNMP polling is active with community name possibilities of *public* and *private*
- WMI monitoring is inactive
- zCollector plugins will collect basic SNMP information, including interface and routing information



- Performance information will only be gathered for interfaces

If auto-discovery for a network is to be deployed, then a mechanism is required to assign discovered devices to a suitable class.

## 2 Automatic device class allocation scenario

Take the scenario where devices are automatically discovered by Zenoss for a particular network. They will be allocated to the /Discovered device class. Note that (at least by default) devices will **only** be discovered if they do respond to ping.

Of the devices that respond to ping, some may support SNMP; others don't. If devices **do** support SNMP then their SNMP Object Identifier (OID) will be collected and stored in Zenoss's Configuration Management Database (CMDB). Further, on a modelling cycle, this OID will be re-checked and will be used to populate the hardware manufacturer and model and the Operating System make and version.

The screenshot shows the Zenoss Core interface for a device at IP 10.191.100.4. The 'Device Status' section shows the device is 'Up' with an availability of 99.615% and an uptime of 497 days, 2 hours, 27 minutes, and 52 seconds. The 'Device Information' section provides details on the hardware (Cisco 7206) and operating system (IOS 12.0(12)).

Component Type	Status
Other	
cpu5min	●
mem5minFree	●
IpRouteEntry	●
IpInterface	●

Figure 7: Status page for a device showing Hardware and OS information

Zenoss has a large database of hardware and software information out-of-the-box, which can be added to and modified by the user.



Figure 8: Product entry for Cisco 7206 router showing SNMP OID association

A Product class also has zProperties which appear to do exactly what is required – assign a device to a device class, based on the SNMP OID.



Figure 9: zProperties for the Product class /Manufacturers/Cisco/7206

Unfortunately with the current version of Zenoss (2.3.2) these zProperties are documented as “For future use” and do not work.

The other alluring possibility for automatically assigning a device to a device class is that the zendisc command documents an **-assign-devclass-script** option. ( This used to be called the **-auto-allocate** option prior to Zenoss 2.2.) Unfortunately, there is no documentation as to how to use this script and the code in `$ZENHOME/Products/DataCollector/zendisc.py` has comments that says this option does not work! So, we have to hand-craft a solution using various facets of Zenoss.

The scenario described here assumes a device to be discovered into the /Discovered class. By default, the device will remain in the /Discovered class; however the mechanism described here means that if a device does not support SNMP on discovery then approximately 1 day can go by before the device will be moved from the /Discovery class to the /Ping class. During this period, the device will be polled for SNMP every 5 minutes and, if no response is received, then the count on the initial event reporting “SNMP agent down”, will be incremented. When the count on the

event gets to 300, the device will be moved to /Ping which, by default, only ping-polls; there are no SNMP polls and no performance templates assigned.

This permits staff to recognise that a newly-discovered device needs an SNMP agent installing and/or configuring for use with Zenoss. Obviously the period elapsed before action is taken, can be adjusted.

If the device is reconfigured **after** this 300x5 minute period, the assumption is that the device will send an SNMP cold start TRAP to the Zenoss system. Zenoss will configure this TRAP to check the current device class of the device and, if it is /Ping, to move the device back to the class /Discovered. The cold start TRAP will also automatically clear any “SNMP agent down” events from the same device.

For newly discovered devices that **do** support SNMP, and devices that may subsequently support SNMP, a script will be run periodically by the Unix cron utility, that uses the device's OID from the Zenoss CMDB and moves the device to an appropriate device class.

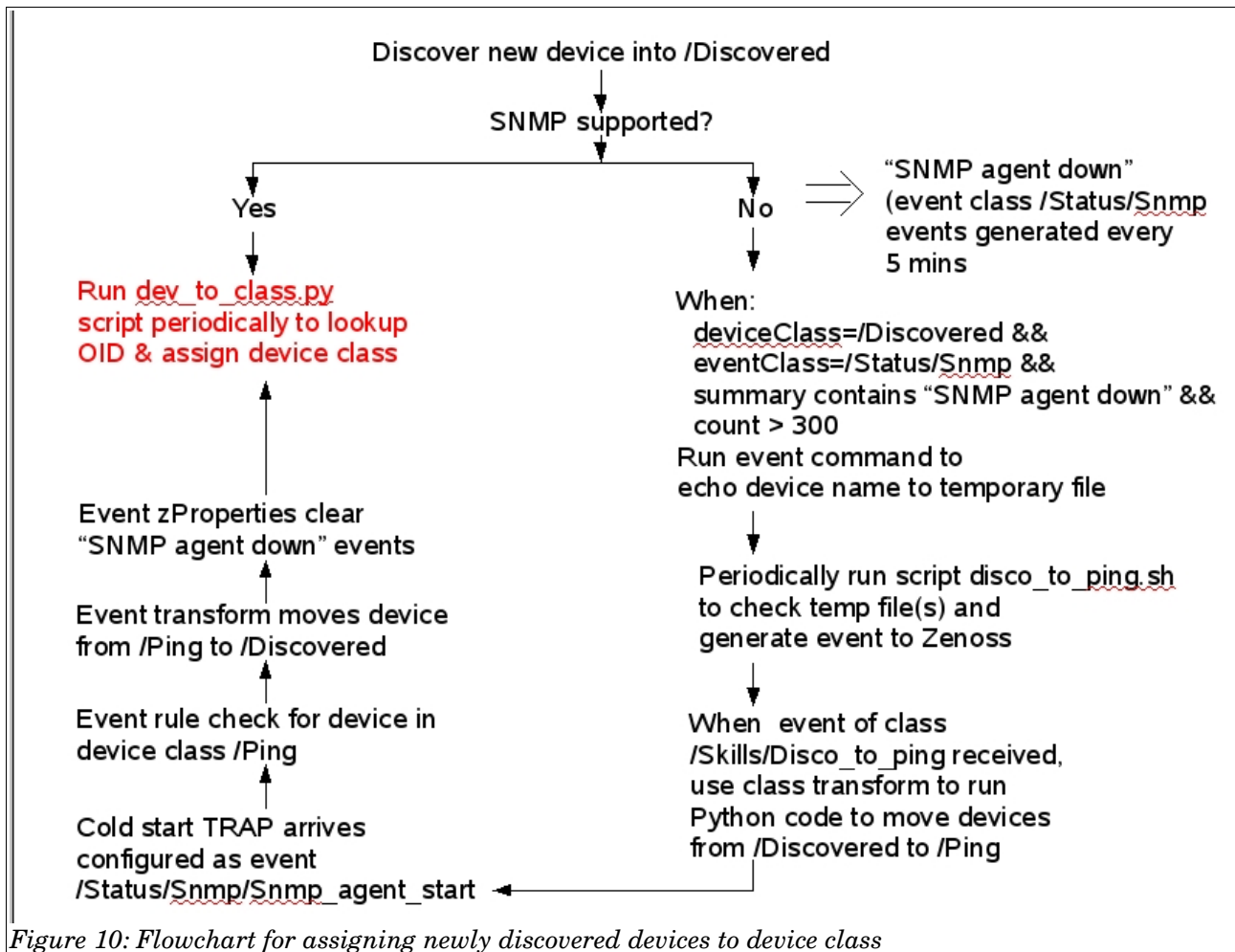


Figure 10: Flowchart for assigning newly discovered devices to device class

## 3 Elements of the solution

This solution uses a number of facilities that Zenoss provides, plus the ability to run scripts periodically using the cron scheduler of the Zenoss operating system.

### 3.1 dev\_to\_class.py scheduled script for device class allocation

The easy, optimal solution is when a device is discovered and responds to SNMP. During the initial discovery, some SNMP information is gathered, including the SNMP OID of the device. Once a device has been discovered with ping, a modelling process takes place. Subsequently, modelling can either be run manually for a device using the drop-down menu and the *Manage -> Model Device* option. Alternatively, the zenmodeler process runs automatically, by default every 12 hours (configure from the left-hand *Collectors -> localhost* menu).

One possibility would appear to be to use the event generated when a device is discovered, to move the device to an appropriate class. Either an event class transform might be considered or an event command. This turns out not to be such a good idea because:

- the event is generated before the collector plugins of the modelling process has been run, so the SNMP OID may not be known at event time
- if an event command is used, commands are actually processed asynchronously every minute (by default) by the zenactions daemon and you can get bad performance issues and inconsistencies in updating Zenoss's Zope CMDB configuration database if several updates run within the 1 minute cycle
- if an event transform is used, similarly performance issues can ensue

Hence, this solution does not attempt to modify a device's class at discovery time but runs a scheduled script which simply checks through all devices currently in the class /Discovered, moves the devices to appropriate classes and then performs a single CMDB commit at the end.

```

jane@bino:~ - Shell - Konsole <3>
Session Edit View Bookmarks Settings Help

#!/usr/bin/env python
# Description: Allocate devices to device classes
# Author: Jane Curry jane.curry@skills-1st.co.uk
# Date: Feb 2nd 2009
# Host system: Open SuSE 10.3
# Zenoss ver: 2.3.2 stack-built
# Updated: Feb
# Comments: From an idea from Cluther ....

import Globals, re, string
from Products.ZenUtils.ZenScriptBase import ZenScriptBase
from transaction import commit
dmd = ZenScriptBase(connect=True).dmd

# Create a dictionary of exact OIDs / device class mappings
deviceClassMap = {
    '.1.3.6.1.4.1.311.1.1.3.1.2': '/Server/Windows',
    '.1.3.6.1.4.1.8072.3.2.10': '/Server/Linux',
    '.1.3.6.1.4.1.9.1.110': '/Network/Router/Cisco'
}

discovered = dmd.Devices.Discovered

# Cycle through all devices in the /Discovered subclass
for device in discovered.getSubDevices():
    #
    # Try for an exact match with auto-discovered HW Product Key
    #
    exact=False
    try:
        deviceClass = deviceClassMap[device.getHWProductKey()]
        discovered.moveDevices(deviceClass, device.id)
        exact=True
        print 'Exact match - moving device %s to class %s' % (device.id, deviceClass)
    except KeyError:
        print 'No exact match for device %s, HWProductKey %s' % (device.id, device.getHWProductKey() )
        pass
    #
    # If no exact match, then how about partial matches?
    #
    if not exact:
        if device.getHWProductKey().startswith('.1.3.6.1.4.1.311'):
            discovered.moveDevices('/Server/Windows', device.id)
            print 'Windows match - moving device %s to class /Server/Windows' %device.id
        elif device.getHWProductKey().startswith('.1.3.6.1.4.1.8072'):
            discovered.moveDevices('/Server/Linux', device.id)
            print 'Linux match - moving device %s to class /Server/Linux' %device.id
        elif device.getHWProductKey().startswith('.1.3.6.1.4.1.9'):
            discovered.moveDevices('/Network', device.id)
            print 'Cisco match - moving device %s to class /Network' %device.id

commit()
"dev_to_class.py" 56 lines --1%--
1,1

```

Figure 11: *dev\_to\_class.py* Python script to move devices from /Discovered to more appropriate classes

Obviously, this script can be modified to add extra exact matches for the SNMP OID and other partial matches based on the start of an OID.

Note that if you have a Zenoss GUI window opened for a device whose class has changed then you will get an error message when you return to that window. You can simply return to any available option in the left-hand menu and continue.

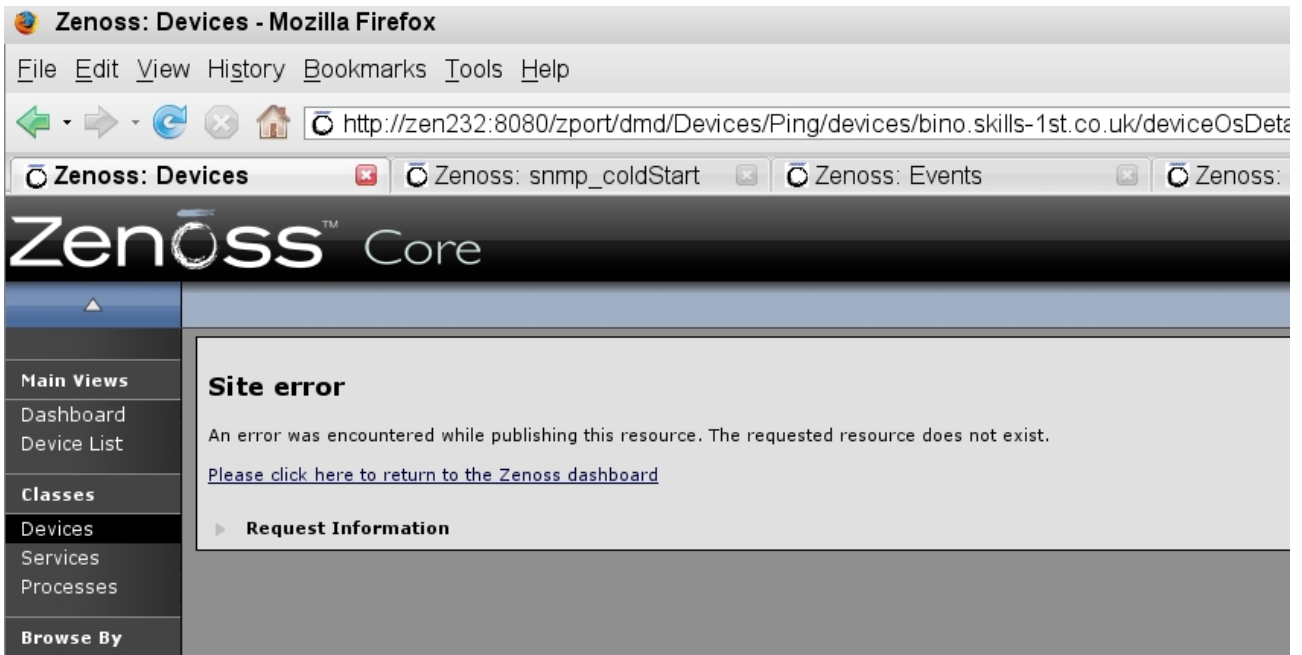


Figure 12: Error message when returning to device page after device class move

### 3.2 Devices that do not (initially) support SNMP

There may be many reasons why Zenoss cannot get SNMP information from a device:

- the device has no SNMP agent
- the device has a different SNMP community name than Zenoss is using
- the device uses a different port for SNMP than Zenoss is using (UDP/161 is the default)
- the device uses a different version of SNMP than Zenoss is using (v1 is the Zenoss default; v2c and v3 are possible)
- there may be a firewall between Zenoss and the device blocking SNMP

Whatever the reason, Zenoss will continue to poll devices in the /Discovered class every 5 minutes. After the first failure, an event appears in the Event Console.

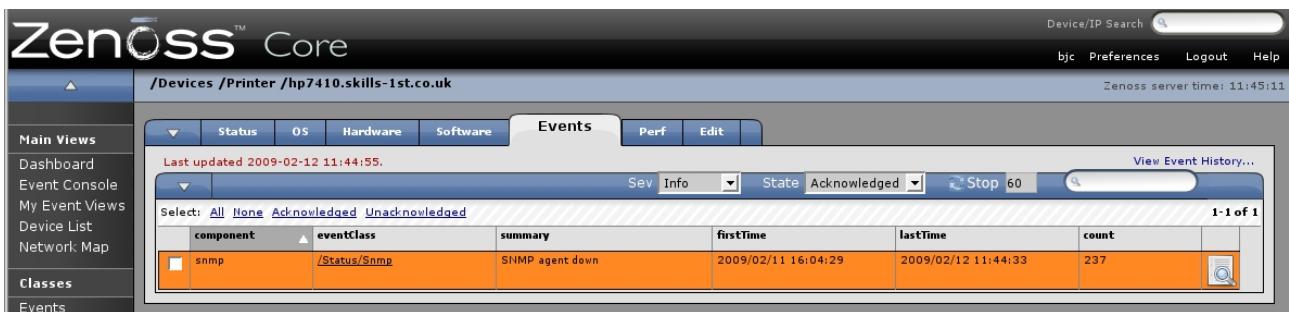


Figure 13: SNMP agent down event with event count increasing

On subsequent 5-minute polls, the event count is increased. This scenario is based on allowing a period after discovery for the SNMP agent to be fixed, so action will be taken when the event count reaches 300 (25 hours). After that, there is no point in continuing to issue SNMP polls to these devices so they need moving to the /Ping device class.

This is achieved using an event command which has a very flexible filter mechanism to define exactly when something should happen. Basically, an event command simply runs a shellscript.

The filter ensures that:

- the device is currently in the class /Discovered – we don't want to affect devices that have already been allocated to useful classes
- the event is of class /Status/Snmp
- the summary of the event contains the string “SNMP agent down”. There are several events that map to the /Status/Snmp event class – we are only interested specifically in the SNMP agent down event
- the count > 300 – obviously this is easily adjusted

When ALL the filters are matched (the criteria are logically ANDed) then the script is run the next time that zenactions wakes up.

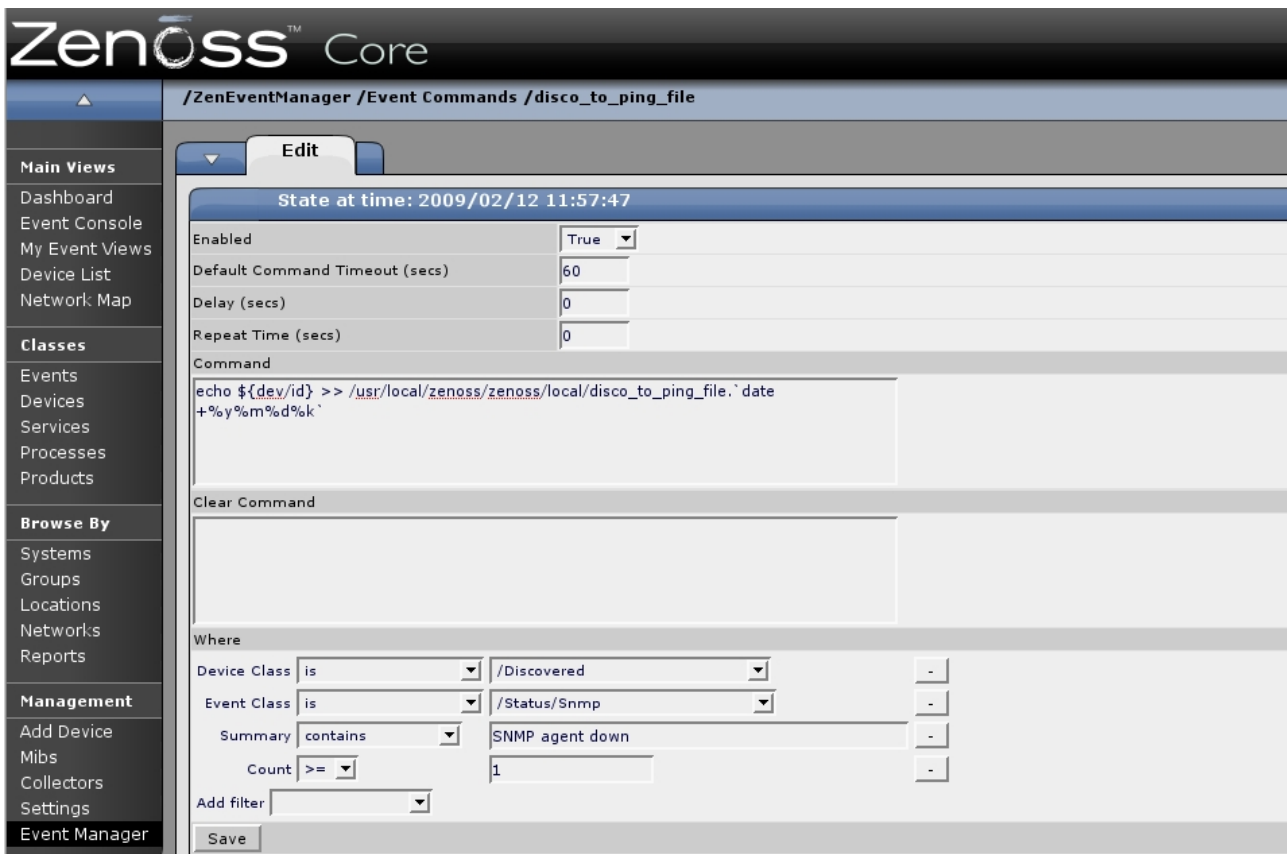


Figure 14: disco\_to\_ping\_file event command

A possible negative aspect of the event command is that it is run asynchronously by the zenactions daemon which runs every minute (by default). If the event command attempts to move a device to a different class and if there are lots of devices that get processed similarly at the same time (as is likely if a network discovery was performed) then you can end up with a large number of event commands all running at once, all trying to modify the CMDB database. When I first tried this, I ended up with 80 concurrent processes, all spawned by zenactions, all trying to update the CMDB. Performance was horrible and the CMDB transactions failed. So, the event command simply echos the name of the device to a temporary file. The screenshot above creates a different file every hour with a <yy><mm><dd><hh> suffix.

Now a mechanism is required to process the temporary data file. A small shellscript is run by the cron scheduler that catenates any temporary data files into a single file, *\$ZENHOME/local/disco-to\_ping.out*. The code that moves devices from one class to another is Python, not shellscript, so rather than call Python code from the script, the Zenoss utility to generate an event, *zensendevent*, is used. This means that there is a tracking event within the Zenoss Event Console for when the process is run.



```
Session Edit View Bookmarks Settings Help
#!/bin/bash
# Source the Zenoss environment
. /usr/local/zenoss/scripts/setenv.sh
# This is the Zenoss server
zenoss=zen232.class.example.org
# Catenate any temporary files into $ZENHOME/local/disco_to_ping.out
# and remove temporary files
cd $ZENHOME/local
cat /dev/null > disco_to_ping.out
for file in `ls disco_to_ping_file.*`
do
    cat $file >> disco_to_ping.out
    rm $file
done
$ZENHOME/bin/zensendevent -d $zenoss -c /Skills/Disco_to_ping -s Info disco_to_ping process run
"disco_to_ping.sh" 19 lines --5%--
```

Figure 15: disco\_to\_ping.sh script run periodically by cron

The zensendevent command takes a number of parameters, including:

- -d the device that generated the event – the Zenoss system in this case
- -c the event class to generate – a locally created class, /Skills/Disco\_to\_ping
- -s the severity of the event – Info (blue) in this case
- The remainder of the line is treated as the event summary

A new event is created, /Skills/Disco\_to\_ping, and it is configured with an event class transform to run the Python code (from an event class, use the drop-down table menu to reach the Transform option). Note that this is an event **class** transform, not an event class **mapping** transform – it runs whenever an event of class /Skills/Disco\_to\_ping arrives.

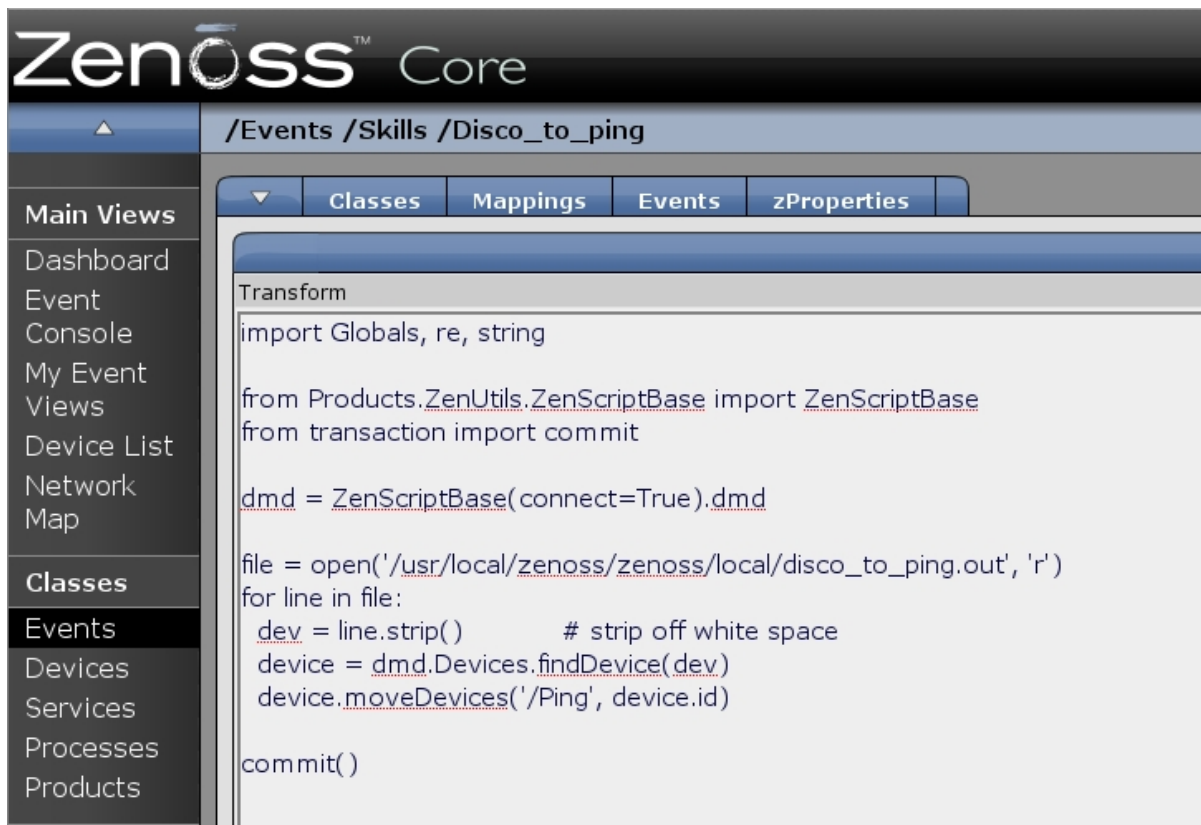


Figure 16: Event class transform for /Skills/Disco\_to\_ping

The script simply works through each line in disco-to\_ping.out and moves each device to the /Ping device class. A single commit is performed at the end which makes for fewer transaction problems with the CMDB database.

Note that you **cannot** run an event transform when the “SNMP agent down” event reaches a certain count as the count field of the event is not available at event transform time.

Thus far, the solution moves devices to an appropriate class soon after discovery if the device supports SNMP and moves non-SNMP devices to the /Ping device class if there is no SNMP support 25 hours after discovery. There are two scripts to be scheduled by cron or run manually – *dev\_to\_class.py* and *disco\_to\_ping.sh*.

### 3.3 SNMP agent installed subsequently for device in /Ping

To get good monitoring for a device, a responsive SNMP agent is a big help. Hopefully, most devices will eventually have an agent installed and configured suitably to communicate with Zenoss. The first indication of a well-configured SNMP agent is likely to be a cold start TRAP from the device to Zenoss.

Zenoss has configuration out-of-the-box that interprets the generic cold start TRAP but it maps to the */Unknown* event class. To effect useful actions, this TRAP needs to map to a specific event. To that end, create a new event subclass, *Snmp\_agent\_start*,

under `/Status/Snmp` (use the SubClasses drop-down table menu and Add New Organizer).

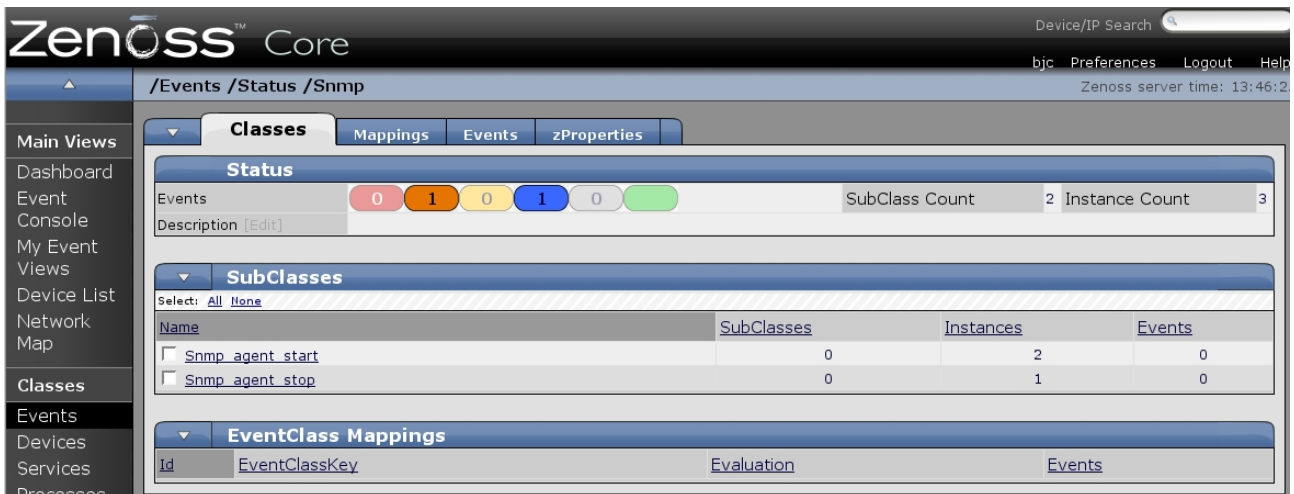


Figure 17: New event subclasses under `/Events/Status/Snmp`

A cold start TRAP in the Event Console can be mapped to this new class very simply by selecting the event and using the drop-down table menu to “Map events to Class” - choose the new `/Status/Snmp/Snmp_agent_start` class from the drop-down list. This mapping will apply to **all** cold start TRAPs.

The scenario here calls for a different action **only** if the device is in the `/Ping` device class – action should certainly not be initiated whenever an SNMP agent is bounced; we are really trying to define the first appearance of an SNMP agent. This is achieved with a second event class **mapping** for the `/Status/Snmp/Snmp_agent_start` event class.

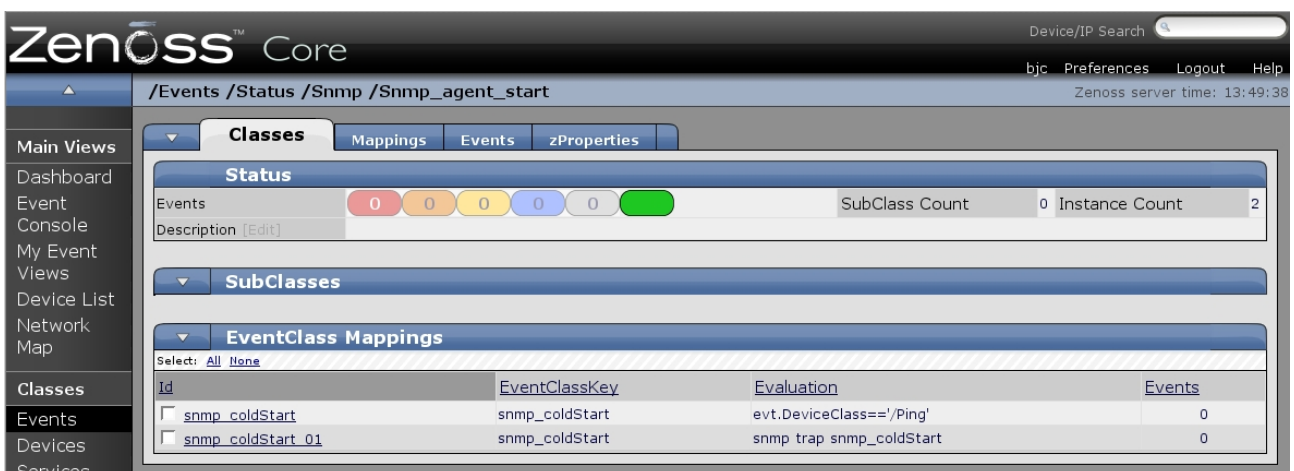


Figure 18: Event class mappings for `/Status/Snmp/Snmp_agent_start`

As can be seen above, the original mapping simply maps if the summary of the event contains the string *snmp trap snmp\_coldStart*. Create a second mapping for this event using the EventClass Mapping drop-down table menu and *Add Mapping*.

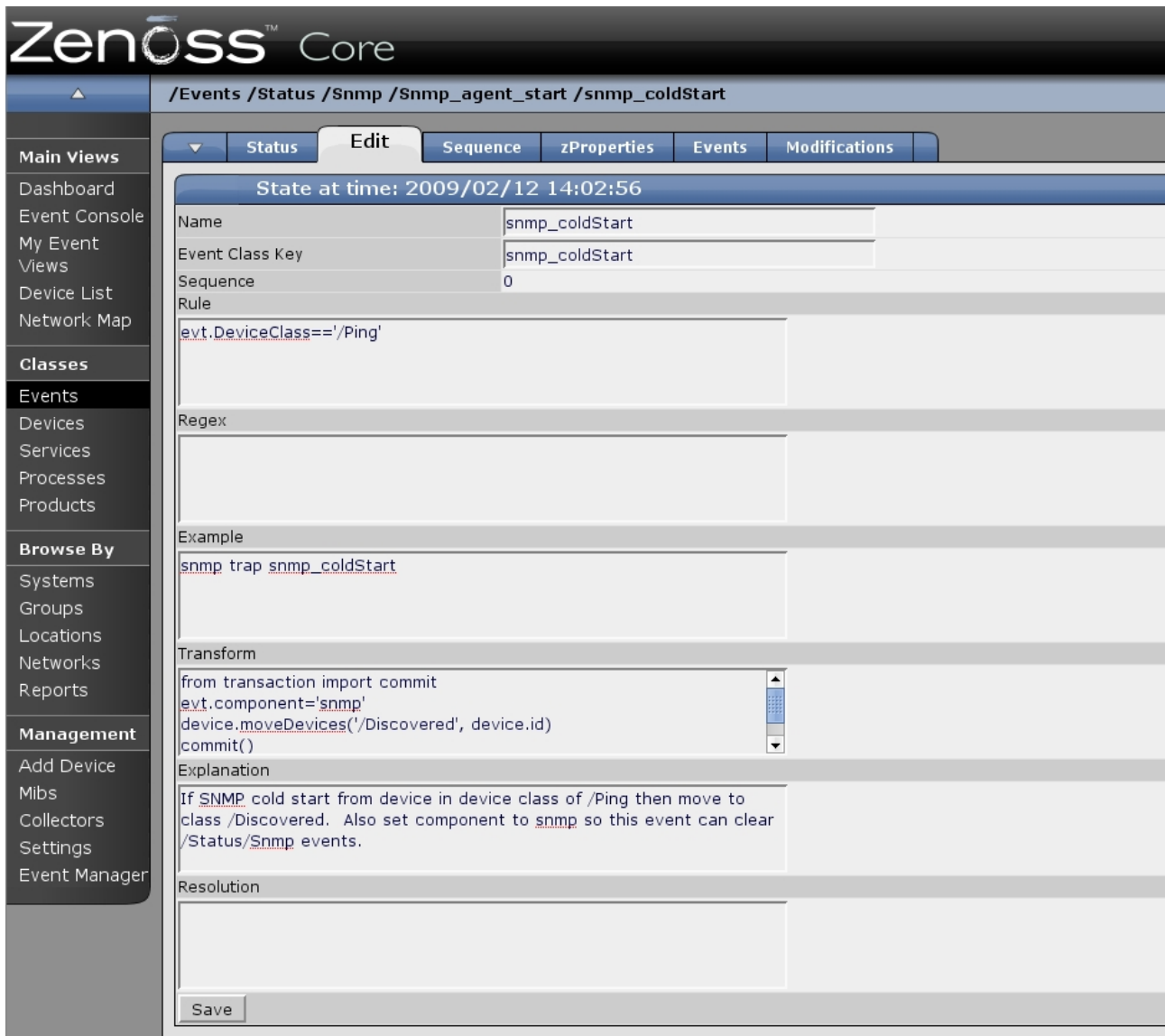


Figure 19: Event class mapping for /Status/Snmp/Snmp\_agent\_start for devices in /Ping device class

This mapping should only match if the device that sent the event is currently in device class /Ping so a mapping **Rule** specifies this. The mapping **Transform** then achieves two things:

- the component field of the event is set to *snmp*. This is so that we can eventually use this event to clear any associated “SNMP agent down” events.
- the device is moved from /Ping to /Discovered

Since there are now two mappings for the `/Status/Snmp/Snmp_agent_start` event class, they should be prioritised using the *Sequence* tab, with the more specific event being sequence number 0 (ie. tested first).

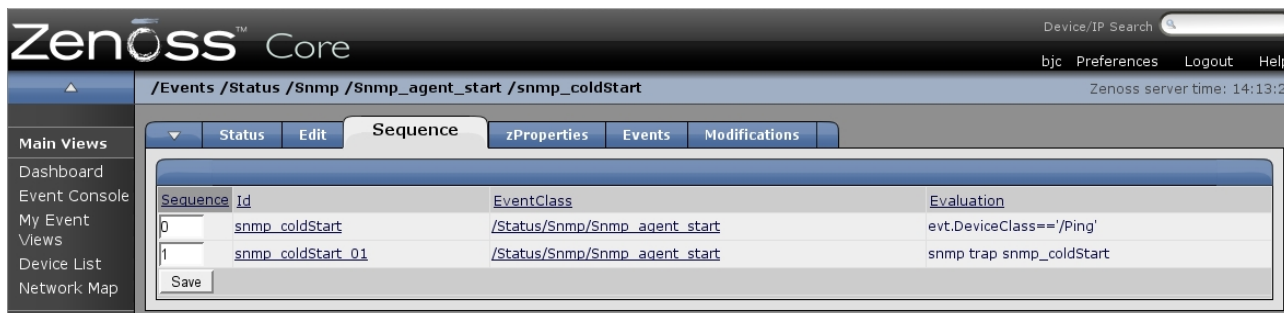


Figure 20: Sequence numbers for the `/Status/Snmp/Snmp_agent_down` event mappings

This ensures that only cold start TRAPs for devices in the `/Ping` class will have their class changed; ordinary reboots of SNMP agents will be unaffected.

To complete the scenario, it would be useful for the specific `/Status/Snmp/Snmp_agent_start` mapping to clear any “SNMP agent down” events. Use the *zProperties* tab to set the *zEventSeverity* to *Clear* and the *zEventClearClasses* to include `/Status/Snmp` events. Note that for *zEventClearClasses* to be utilised, events must be from the same **device** and the event **component** field must be the same. This is why the event mapping transform sets the component field to 'snmp' to match those generated by the “SNMP agent down” events.

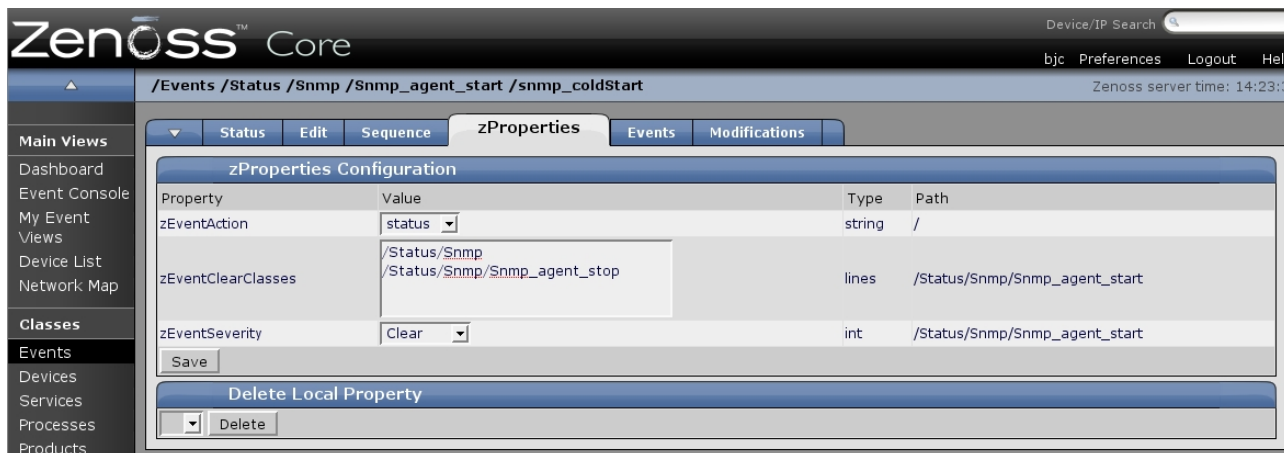


Figure 21: zProperties for event class mapping `/Status/Snmp/Snmp_agent_start`

At this point, recently discovered devices which have latterly had an SNMP agent installed, are now in the same position as those devices who had SNMP support on discovery so the periodic `dev_to_class.py` script should move the devices to a more appropriate device class, based on their SNMP OID.

## 4 Conclusions

This paper demonstrates a method for classifying devices that are automatically discovered. For small environments where devices can be added and / or configured manually, it is overkill. For larger environments where hundreds of devices may be discovered, especially where a significant proportion do not initially support SNMP, the solution seems to be helpful.

When testing the solution, *\$ZENHOME/log/zenactions.log* is useful for checking the progress of event commands and *\$ZENHOME/log/zenhub.log* is useful for debugging problems with event transforms. Python code can be tested as a standalone program and bits of python can be tested using Zenoss's *zendmd* command environment.