Zenoss Event Management

Version 3

September 2009

Updated January 2010

Jane Curry

Skills 1st Ltd

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

Synopsis

This paper is intended as an intermediate-level discussion of the Zenoss event system. It assumes that the reader is already familiar with the Zenoss Event Console and with basic navigation around the Zenoss Graphical User Interface (GUI). It looks in some detail at the architecture behind the Zenoss event system – the daemons and how they are interrelated – and it looks at the structure of a Zenoss event and the event life cycle.

Zenoss can receive events from many sources in addition to Zenoss itself. Events from Windows, Unix syslogs and Simple Networks Management Protocol (SNMP) TRAPs are all examined in detail.

The process by which an incoming event is transformed into a particular Zenoss event is known as event mapping and has a number of different possible techniques for performing that conversion. These will all be explored along with the creation of new event classes.

Once an event has been received and classified by Zenoss, automation may be required. Alerting by email and pager are discussed as is the ability to run any script as an Event Command.

This paper was written using Zenoss 2.4.1.

The paper is a companion text to the Zenoss Event Management Workshop.

Notations

Throughout this paper, text to by typed, file names and menu options to be selected, are highlighted by *italics;* important points to take note of are shown in **bold.**

Table of Contents

1 Introduction

Zenoss is an Open Source, multi-function systems and network management tool. There is a free, Core offering (which does seem to have most things you need), and a chargeable offering, Enterprise, which has extra add-on goodies such as high availability configurations, distributed management servers, role-based access and various support contracts which include some education and consultancy. For a comparison of the "fee" alternatives, try http://www.zenoss.com/product/pricing .

Zenoss offers configuration discovery, including layer 3 topology maps, availability monitoring, problem management and performance management. It is designed around the ITIL concept of a Configuration Management Database (CMDB), "the Zenoss Standard Model". Zenoss is built using the Python-based Zope web application server and uses the object-oriented Zope Object Database (ZODB) as the CMDB, used to store Python objects and their states. Zenoss uses ZEO, as a layer between Zope and the ZODB.

The relational MySQL database is used to hold current and historical events. Performance data is held in Round Robin Database (RRD) files.

The default protocols for monitoring are typically "agentless" - the Simple Network Management protocol (SNMP), Windows Management Instrumentation (WMI) and collecting events from syslogs. It is also possible to monitor devices using telnet, ssh and to use Nagios plugins.

Zenoss provides a good "Getting Started with Zenoss" document along with a "Zenoss Administration Guide" and a "Zenoss Developer's Guide"; you can get these from http://www.zenoss.com/community/docs . There is also a wealth of information on the Zenoss website but it is rather diffused between FAQs, HowTos, a Wiki and contributions to the various forums. A useful book is available from PACKT Publishing, "Zenoss Core Network and System Monitoring" by Michael Badger, which provides much of the same information as the Zenoss Administration Guide but in a much clearer format with plenty of screenshots.

This paper is an attempt to expand on the event information in the Administration Guide by drawing on my own experience and the collected wisdom of the community contributions.

2 Zenoss event architecture

2.1 Event Console

When an event arrives at Zenoss, it is parsed, associated with an event classification and then typically (but not always), it is inserted into the **status** table of the **events**

database. Events can then be viewed by users using the Event Console of the Zenoss Graphical User Interface (GUI).

There are three ways to access the Event Console. The main Event Console is reached from the *Event Console* menu on the left. The default is to show all status events with a severity of Info or higher, sorted first by severity and then by time (most recent first). Events are assigned different severities:

- Critical Red
- Error Orange
- Warning Yellow
- Info Blue
- Debug Grey
- Clear Green

The events system has the concept of active status events and historical events (two different database tables in the MySQL events database).

Events in the console can be filtered by Severity (Info and above by default) and by State - New, Acknowledged and Suppressed - where New and Acknowledged are shown by default. Any event which has been Acknowledged changes to a wishy-washy version of the same colour. A Suppressed event also has the wishy-washy version of the colour. There is a Search box at the top right for filtering events based on the presence of any string within any field of an event.

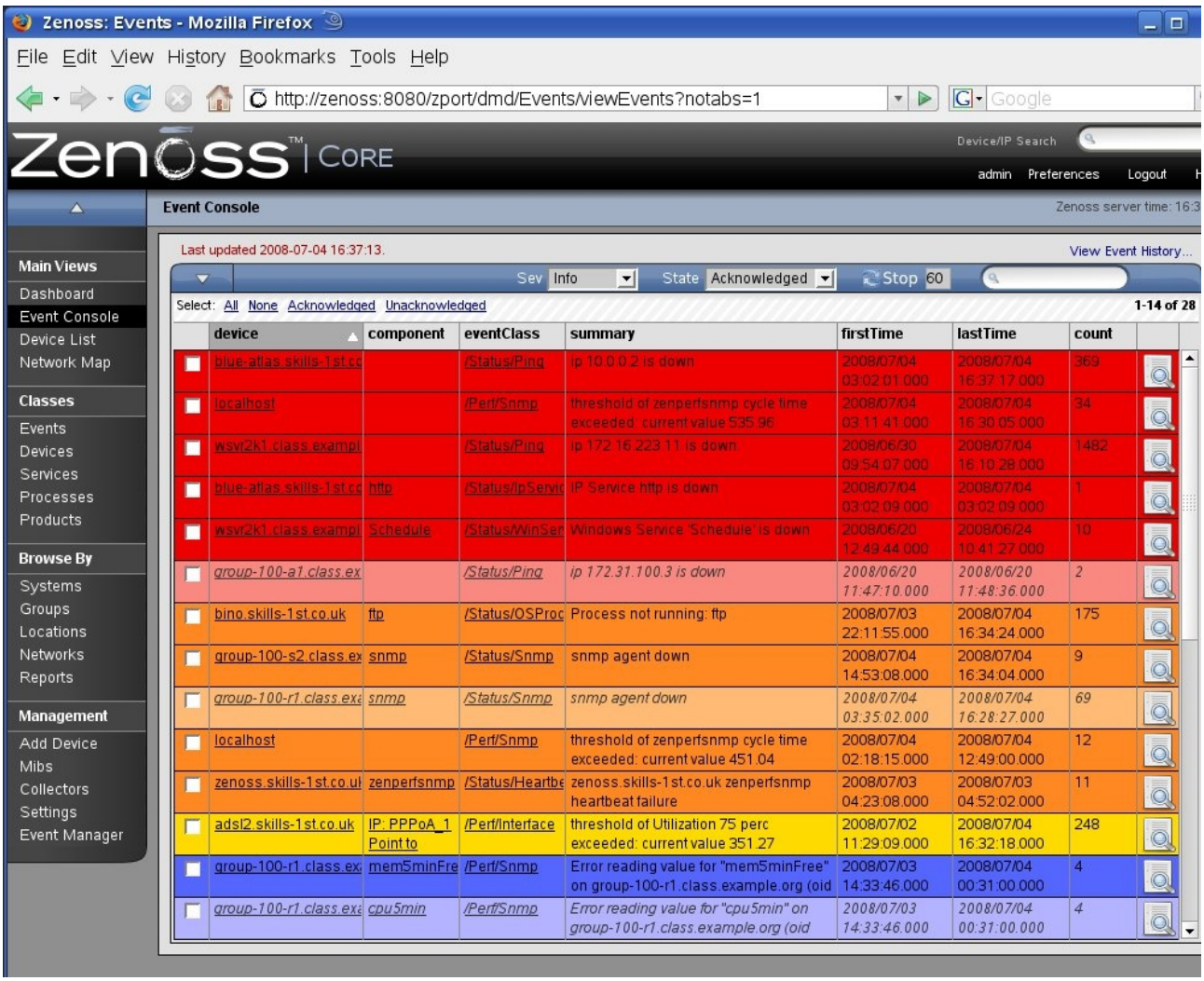

Figure 1: Zenoss Event Console

From the Event Console, one or more events can be selected by checking the box alongside the event and the table menu dropdown can be used for various functions including “Acknowledge Events”, “Move to History” and “Map Events to Class”.

The column headers of the Event Console can be used to change the sorting criteria and the icon at the far right of the event can be used to display the detailed data of the event.

The detailed data shows the default event fields under the *Fields* tab, any user-defined fields under the *Details* tab and the *Log* tab records actions such as acknowledge and clearing, along with date, time and the user that performed these actions. It is also possible to add your own user messages to en event's Log but only while it is in the status table of the events database, not once it has been moved to the history table (Ack / UnAck status makes no difference).

Field	Value
dedupid	zenoss.skills-1st.co.uk|sshd|||5|PAM audit_log_acct_message() failed: Operation not permitted
evid	0a00008337aba1e7e368ce5
device	zenoss.skills-1st.co.uk
component	sshd
eventClass	/Unknown
eventKey	
summary	PAM audit_log_acct_message() failed: Operation not permitted
message	PAM audit_log_acct_message() failed: Operation not permitted
severity	5
eventState	1
eventClassKey	sshd
eventGroup	syslog
stateChange	2009/01/14 16:05:04.000
firstTime	2009/01/07 11:30:47.000
lastTime	2009/01/14 09:06:01.000
count	2
prodState	1000
suppid	
manager	localhost
agent	zensyslog
DeviceClass	/Server/Linux
Location	/Cedar_Chase
Systems	|
DeviceGroups	|
ipAddress	10.0.0.131
facility	authpriv
priority	2
ntevid	0
ownerid	admin
clearid	
DevicePriority	3
eventClassMapping	
monitor	localhost

Figure 2: Detailed data for an event - default Fields tab

The fields under the *Details* tab may include text messages for the *Explanation* and *Resolution* event fields.

Note that if you wish to Undelete an event from the history tables of the database, this is possible with the dropdown table menu; however it does not change any previous Acknowledged status.

By default, the Event Console is refreshed every 60 seconds. If you wish to freeze the console, click the "Stop" link at the top of the console (beside the box with 60 in it); the link changes to say "Start". To return to automatic refresh, click the "Start" link.

An Event Console can also be accessed which automatically filters events for a particular device. Navigate to the main page for a device and use the *Events* tab to show all events for that device.

The third method of displaying an Event Console applies an automatic filter of event class or subclass. Start from the *Events* menu on the left. The *Events* tab will show an Event Console filtered by the chosen class or subclass. Note that the top-level dropdown menu has an option to *Add Event*. This is useful for generating test events.

Each of these three methods of accessing an Event Console shows **active** events. To see events that have been sent to the history table of the events database, click the blue *View Event History* link at the top right of the Event Console.

2.2 Event database tables and the Event Manager

Zenoss events are held in a MySQL database called **events** which is created when Zenoss is installed. By default, the *zenoss* user can access this database with a password of *zenoss*.

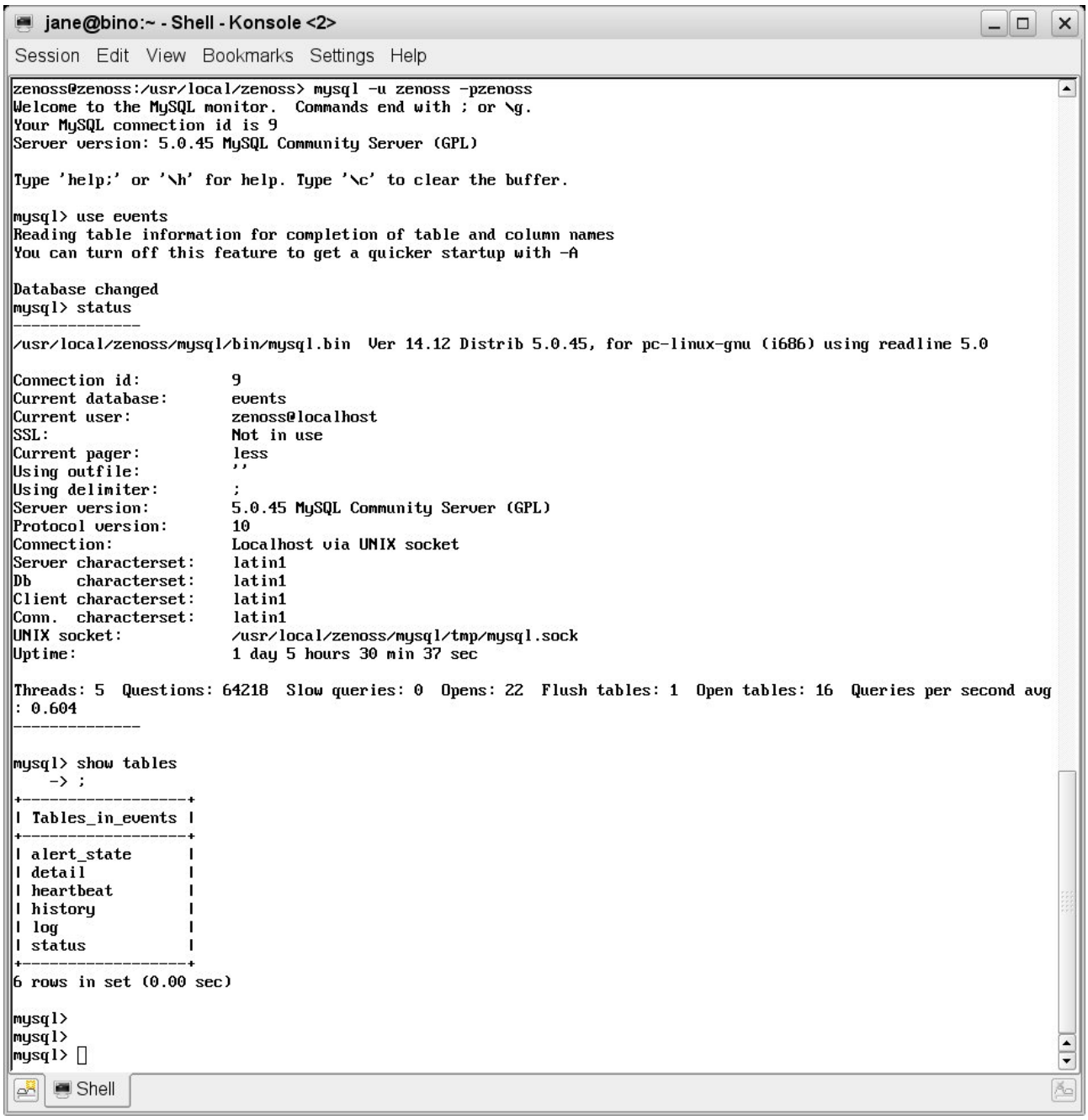

Figure 3: Using MySQL to examine the events database

The main tables within the events database are **status** and **history**. The active events are kept in the status table and the historical events (typically resolved, cleared events) are held in the history table. The format of each of these tables and the valid fields for a Zenoss event can be seen be examining the database setup file in */usr/local/zenoss/zenoss/Products/ZenEvents/db/zenevents.sql* .

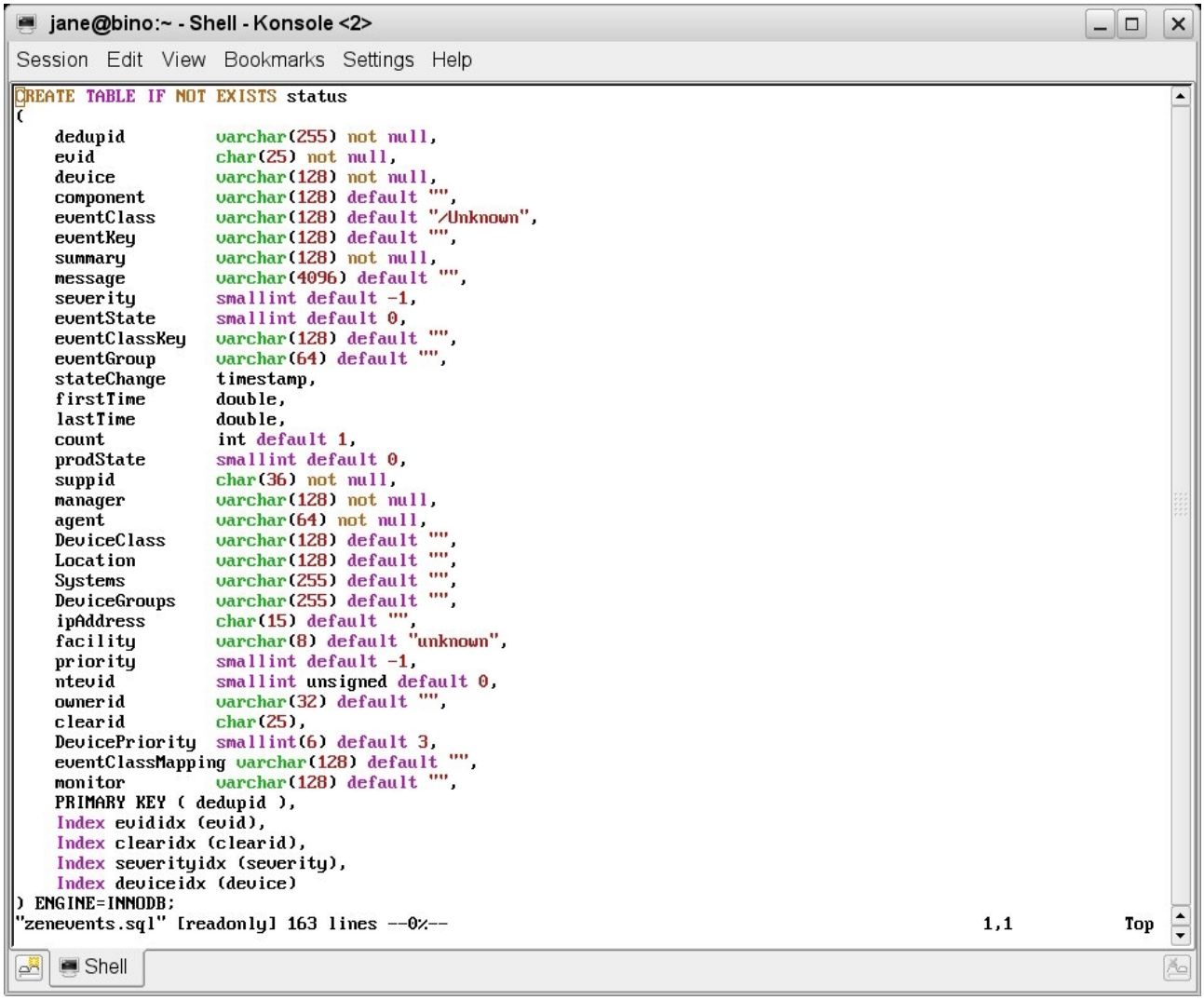

Figure 4: Definition of status event fields in zenevents.sql

zenevents.sql also defines the **history** table in a similar fashion.

A further four tables are defined for **heartbeat, alert_state, log** and **detail** . The detail table can be used to extend the default event fields to include any information that the Zenoss administrator requires for an event, including Explanation and Resolution text.

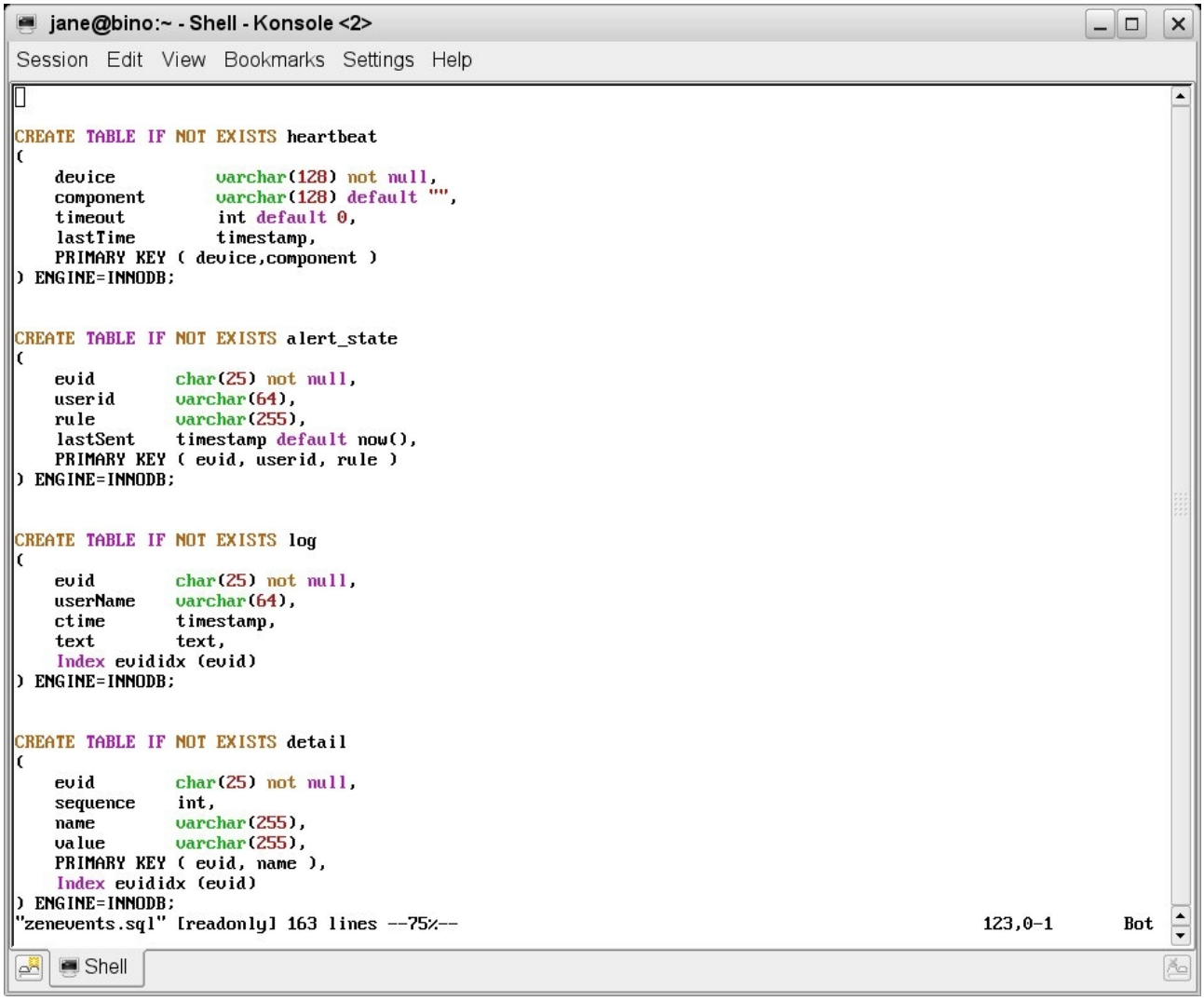

Figure 5: zenevents.sql showing heartbeat, alert_state, log and detail tables

Some of these event fields are particularly pertinent depending on how the event was generated:

- Syslog events populate the **facility** and **priority** fields
- Windows events populate the **ntevid** field
- SNMP TRAPs populate at least a **community** field in the detail table. They also use the detail table to provide any variables passed by an SNMP TRAP.
- The **agent** field denotes which Zenoss daemon generated or processed the incoming event; for example, zentrap, zeneventlog, zenping .

Connection information for the events database along with caching and maintenance parameters, can be accessed from *Event Manager* in the menu on the left of the Zenoss console.

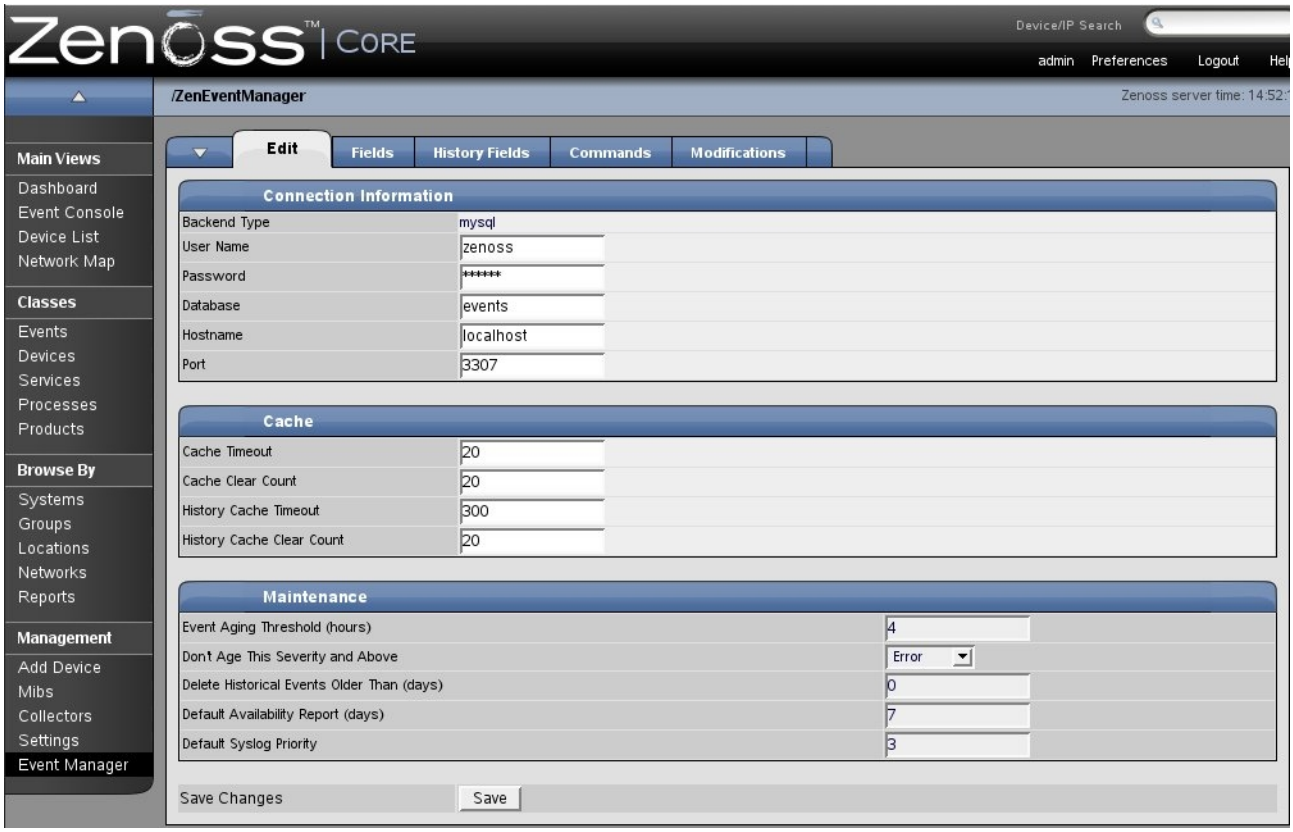

Figure 6: Event Manager options for MySQL events database

By default, status events of severity below Error, are aged out to the events history table after 4 hours. Historical events are never deleted.

2.3 Event life cycle

The life cycle of an event has eight phases:

- Event generation
- Device context – additional information about the device that generated the event
- Event class mapping – to distinguish one type (class) of event from another
- Event context - additional information pertinent to a class of event
- Event transform – manipulation of event fields
- Database insertion
- Resolution
- Age out

Processing of an event depends on the **event class** that an event is assigned to – the value of its **eventClass** field. A description of each of these phases will be given here: subsequent sections of the paper provide more details of some areas.

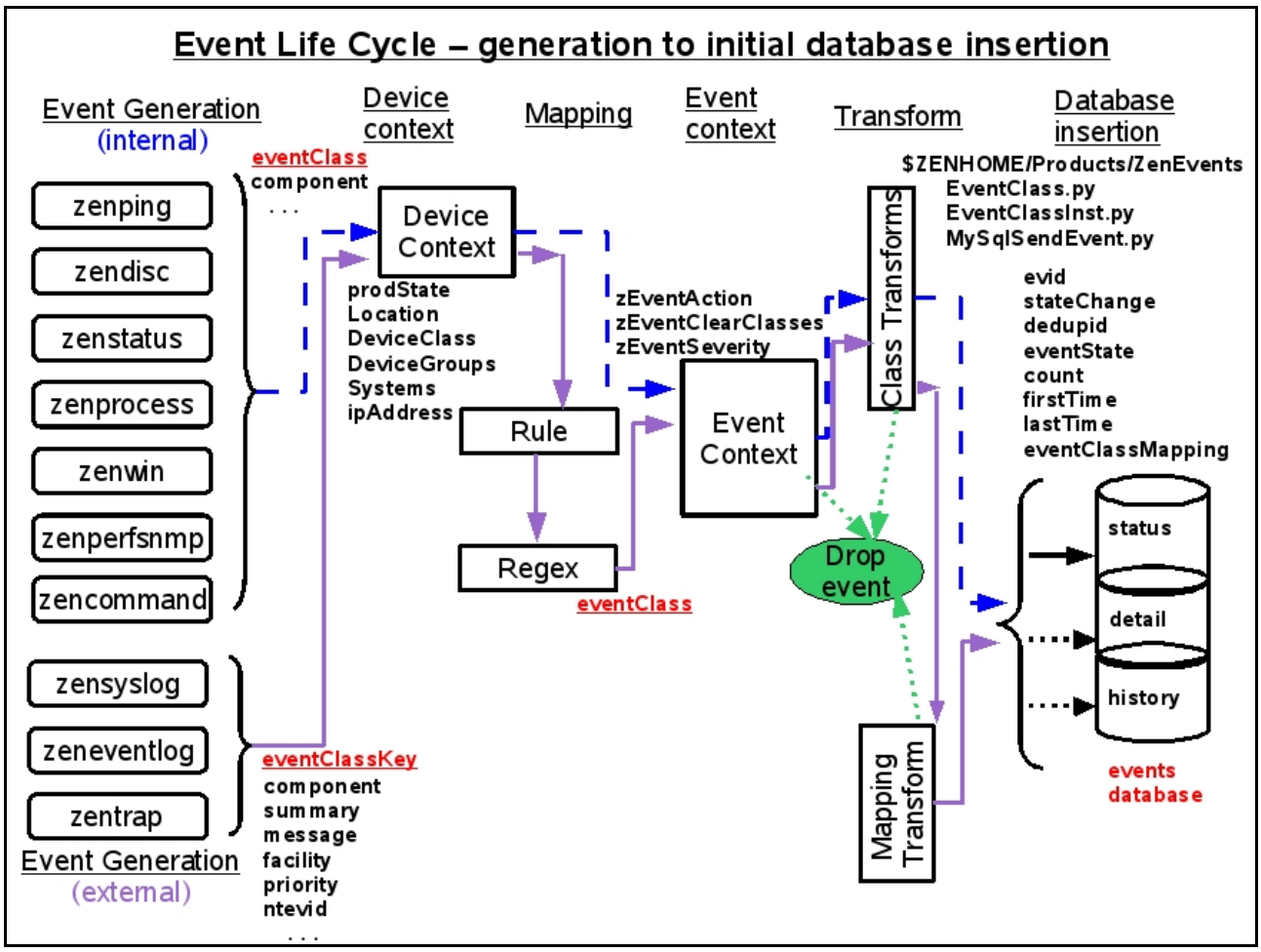

Figure 7: Event life cycle, generation to database insertion

In Figure 7, the first six phases of the event life cycle are shown. The blue, dashed path shows the progress of an internally generated Zenoss event, which does not pass through an event mapping phase. An **eventClass** field is produced by the daemon that generated the event. Its only way to apply a transform is as a class transform.

The purple path shows the progress of an event that is generated externally to Zenoss. The initial parsing daemon must provide an **eventClassKey** field which is then used, along with other fields, in an event class mapping Rule and/or Regex, which in turn provides an **eventClass** field. After mapping, the event may pass through both an event class transform and an event mapping transform.

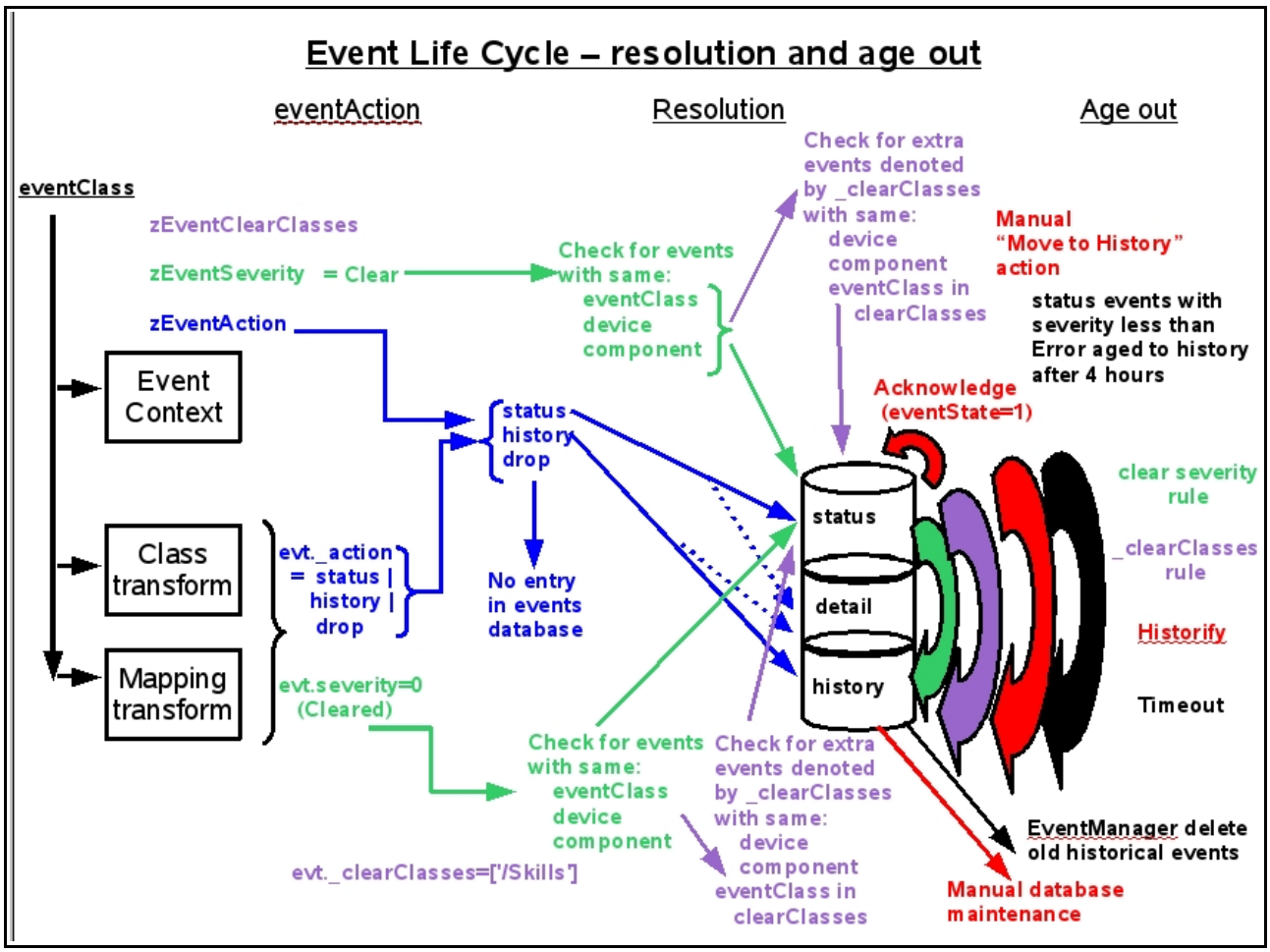

Figure 8: Event life cycle, resolution to age out

Figure 8 shows the latter phases of the event life cycle. Blue paths show the effect of the zProperty zEventAction on the insertion of the event into the various tables of the event database. Green paths show the effect when the zProperty zEventSeverity is equal to *Cleared*. Purple paths show the additional clearing effect of the zProperty zEventClearClasses.

The movement of events between tables of the events database are colour-coded with green for clearing based on severity=*Cleared*, purple for clearing based on zEventClearClasses, red for manual actions (including Acknowledge which does not move the event between tables), and black for movements based on timeouts.

2.3.1 Event generation

Fundamentally, events will either be generated by Zenoss itself in the process of discovery, availability and performance checking, or events will be generated outside Zenoss and captured by specialised Zenoss daemons.

Zenoss daemon	Example of when event generated
zenping	ping failure on interface
zendisc	new device discovered
zenstatus	TCP / UDP service unavailable
zenprocess	process unavailable
zenwin	Windows service failed
zenperfsnmp	SNMP performance data collection failure

Table 2.1.: Events generated by Zenoss itself

Zenoss daemon	Example of when event generated
zensyslog	processes syslog events received on UDP/514 (default)
zeneventlog	processes Windows events received using WMI
zentrap	processes SNMP TRAPs received on UDP/162

Table 2.2.: External events captured by specialised Zenoss daemons

Events generated internally by Zenoss need no further processing to interpret the event. The daemon that generates the event parses the native information and assigns a value to the **eventClass** field and any other relevant fields such as **component, summary, message** and **agent**. Typically the **eventClassKey** field will be blank. Some Zenoss daemons populate the **eventKey** field (for example an Interface discovery event will populate the eventKey field with the IP address of the discovered interface).

Events that are initially generated outside Zenoss are captured by **zensyslog**, **zeneventlog** or **zentrap**. These daemons each have a parsing mechanism to interpret the native event into the Zenoss event format. The Python code for this parsing is in */$ZENHOME/Products/ZenEvents*. (By default, $ZENHOME will be */usr/local/zenoss/zenoss*). *SyslogProcessing.py* decodes syslog events; *zentrap.py* decodes SNMP TRAPs.

Typically, these parsing mechanisms do not deliver a value for **eventClass**; rather they deliver a value for the **eventClassKey** field, along with values for some other fields such as component, summary, message and agent. It is then the job of the event mapping phase to distinguish the event class.

2.3.2 Application of device context

Early in the event processing life cycle, **device context** is applied to the event. This means that six fields of the event are populated by determining the device that

generated the event and then looking up the following values for the device in the ZODB database:

- prodState
- Location
- DeviceClass
- DeviceGroups
- Systems
- ipAddress (may have already been assigned)

2.3.3 Event class mapping

Event class mapping tends only to be applicable to events that originate outside the Zenoss system. It is the process by which an event is assigned a value for its **eventClass** field and, potentially, other fields.

Typically, the event generation phase will deliver an event with a few fields populated; generally this does **not** include the eventClass field but **does** include the eventClassKey field. Often the Zenoss parsing daemon (such as zensyslog), will use the **same** eventClassKey for several different native events. For example, an eventClassKey of *dropbear* is used for several login security events. The component, summary, message and agent fields may also be populated.

The event class mapping phase examines the event (such as it is, so far) and then uses a number of tests to determine the eventClass to assign to this event:

1. An eventClassKey field **must** exist for mapping to be successful.
2. A Python **Rule** can be written to test any available field of the event or any available attribute of the device from which the event came. Such rules can be complex Python expressions, including logical ANDs and ORs. If the rule is satisfied, the incoming event's eventClass field is given the class associated with that mapping. If the rule is not satisfied, this mapping is discarded, the class is not associated, and the next mapping will be tested for a match. A Rule does **not** have to exist in a mapping instance.
3. If the Rule is satisfied (or does not exist), the mapping can then use a **Regex** Python regular expression to parse the event's summary field, checking for particular strings. The Regex can also assign parts of the summary field to new, user-defined detail fields of the event. If a Rule exists and is satisfied, the class mapping **will** apply, even if the Regex is **not** satisfied; any user-defined fields in the Regex will **not** be created if the Regex does not match. If a Rule does **not** exist then the Regex **must** be satisfied for the mapping (and any transform) to apply.

4. The GUI dialogue that defines the mapping specifies the eventClassKey, the Rule, the Regex and any Transform. A **sequence number** is also available so that if multiple incoming events have the same eventClassKey then the sequence number defines the order in which the various mappings will be applied, lowest number first. The first Rule / Regex mapping combination that matches will be applied.

2.3.4 Application of event context

Event context is defined by the zProperties of an event. Event context can be defined at the event class level, for an event subclass, or at the event mapping level. As with all object-oriented attributes, the values are inherited by child objects so applying event context to a class automatically sets it for any subclasses and subclass mappings. The three event context attributes are:

- zEventAction status | history | drop default is status
- zEventClearClasses by default this is an empty Python list of strings
- zEventSeverity *Original* by default

Event context is applied in the event life cycle, after Rule and Regex processing but before any event transforms. Thus, the zEventAction zProperty can specify the history database but an event transform could override that action by setting the evt._action value to status.

2.3.5 Event transforms

Event transforms can be specified for an event class mapping or for an event class (or subclass). A transform is written in Python and can be used to modify any available fields of either the event or the device that generated the event. It can also create user-defined fields.

From Zenoss 2.4, cascading event transforms mean that class transforms are applied from **every** level in the appropriate class hierarchy, followed by any transform for an applied event mapping. Prior to Zenoss 2.4, **either** a mapping transform was applied, **or** a class transform, but not both. Class transforms were only applied to the exact class, not from the event class hierarchy.

A transform in an event mapping will only be executed once the eventClassKey has been matched, and the Rule has been satisfied (if it exists). If a Rule does not exist, any Regex has to be satisfied for the transform to be executed.

2.3.6 Database insertions

Zenoss events are stored in a MySQL database called **events** (by default). The events databases has a number of tables defined – see */usr/local/zenoss/zenoss/Products/ZenEvents/db/zenevents.sql* for the configuration of tables and triggers for the events database.

The main tables for the event life cycle are the **status** table for active events, the **history** table for resolved events and the **detail** table for user-defined fields of events.

Some fields of the event are only assigned at database insertion time – they are not available at event mapping or event transform time. These include:

- count
- evid
- stateChange
- dedupid
- suppid
- eventClassMapping
- firstTime is the same as lastTime until database insertion

The Python code that drives database insertion can be found in *$ZENHOME/Products/ZenEvents;* E*ventClass.py, EventClassInst.py* and *MySqlSendEvent.py* are the main files. Some of the event fields can only be determined by reference to other events already in the database – such as count, dedupid, stateChange and firstTime.

Zenoss automatically applies a duplication detection rule so that if a "duplicate" event arrives, then the repeat count of an existing event will simply be incremented. "duplicate" is defined as having the following fields the same:

- device
- component
- eventClass
- eventKey
- severity

If the event does not populate the eventKey field, then the summary field must also match. At database insertion time, the **dedupid** field is created by concatenating the above fields together, separated by the pipe (vertical bar) symbol. Thus an example dedupid might be:

```
zenoss.skills-1st.co.uk|su|/Security/Su||5|FAILED SU (to root)jane on /dev/pts/1
```

where the device is *zenoss.skills-1st.co.uk*, component is *Security*, eventClass is */Security/Su* , the eventKey is unset, severity is *5* (Critical), and the summary is *FAILED SU (to root) jane on /dev/pts/1* .

2.3.7 Resolution

Resolution is generally the process by which an event is moved from the status table of the events database to the history table.

A second possible definition of "resolution" is that the event is dropped entirely, never reaching the status table – this can be achieved either from the event context by setting zEventAction to *drop*, or in a transform with *evt._action=drop* . These same mechanisms can also be used during initial event processing, to set the event action to *history*, thus preventing the event from ever appearing in the status table.

A third possibility is to set the event's eventState field to *Suppressed* which means the event does not display, by default, in the Event Console status display. The suppressed eventState mechanism only appears to be used by the *zenping* daemon.

An event can be "resolved" by human intervention. The Event Console dropdown table menu provides an option to *Move to history*; the Zenoss Administration Guide describes this as "historifying". It also provides the option to *Acknowledge* an event. Acknowledging changes the eventState field to *Acknowledged* (as opposed to *New);* it changes the colour of the event to a wishy-washy version of the same colour. It does **not** move the event into the history table.

The more interesting forms of event resolution involve correlation of events; there are two different mechanisms. The basic principle is that "good news" clears "bad news".

The first clearing mechanism is that any event with a severity of **Cleared** will search the status table of the events database and move any similar events to the history table. "Similar" is defined as having the same eventClass, device and component fields. All "similar" events are cleared, not just the most recent.

When correlation takes place a number of the existing "bad news" event fields are updated. stateChange, deletedTime and clearid are all modified with clearid becoming the value of the evid field of the clearing, "good news" event. The "good news" event, with its *Cleared* severity, is automatically moved to the history table.

The second correlation mechanism is to specify in the event context, one or more event classes in the **zEventClearClasses** zProperty. This attribute will only be used on **clearing** events. The effect is that any similar events of the listed classes will **also** be cleared, in addition to events of the same class as the clearing event. "Similar" in this case means the same **device** and **component** fields, plus the class specified in **zEventClearClasses**. Note that the same effect can be achieved in a transform by assigning a list of class names to *evt._clearClasses* .

2.3.8 Ageing out events

Maintenance is require on the tables of the events database or the disk will simply fill up eventually. Two mechanisms are provided by the Event Manager:

- By default, events with severity less than Error will be aged from the status table to the history table after 4 hours. These parameters can be modified.
- Historical events can be deleted once they are older than a given number of days. The default is 0 – that is, no events are deleted from the history table automatically.

Manual maintenance on the MySQL database may also be required. Zenoss provides a utility in $ZENHOME/Products/ZenUtils:

`ZenDeleteHistory.py --numDays=10` to clear history events older than 10 days

The script should be run as the zenoss user.

Alternatively, the MySQL database can be manipulated directly by those with SQL knowledge. Here are a few examples; again, work as the zenoss user (note you need the trailing semicolon on SQL statements):

`mysql -uzenoss -pzenoss -Devents` Access mysql as zenoss user, to events db

`select * from heartbeat;` Show all heartbeat events

`delete from heartbeat where device='localhost';` Delete local heartbeat events

`select * from status where device='mybox';` Show all status events for mybox

`select * from detail where evid NOT IN (select evid from status UNION select evid from history);` Show detail records for events deleted from status and history

`delete from detail where evid NOT IN (select evid from status UNION select evid from history);` Delete detail records for events deleted from status and history

3 Events generated by Zenoss

In the course of discovery, availability monitoring and performance monitoring, Zenoss may generate events to represent a change in the current status. Although many events are “bad news” it should be recognised that events can also be “good news” - Interface Up, Threshold no longer breached, etc.

Events generated by Zenoss are dependent on the various polling intervals configured. To examine the default parameters, use the *Collectors* option from the Zenoss left-hand menu. Click on *localhost* (the collector on the Zenoss system). **Note** that earlier versions of Zenoss used the term and menu-option M**onitors** rather than C**ollectors**.

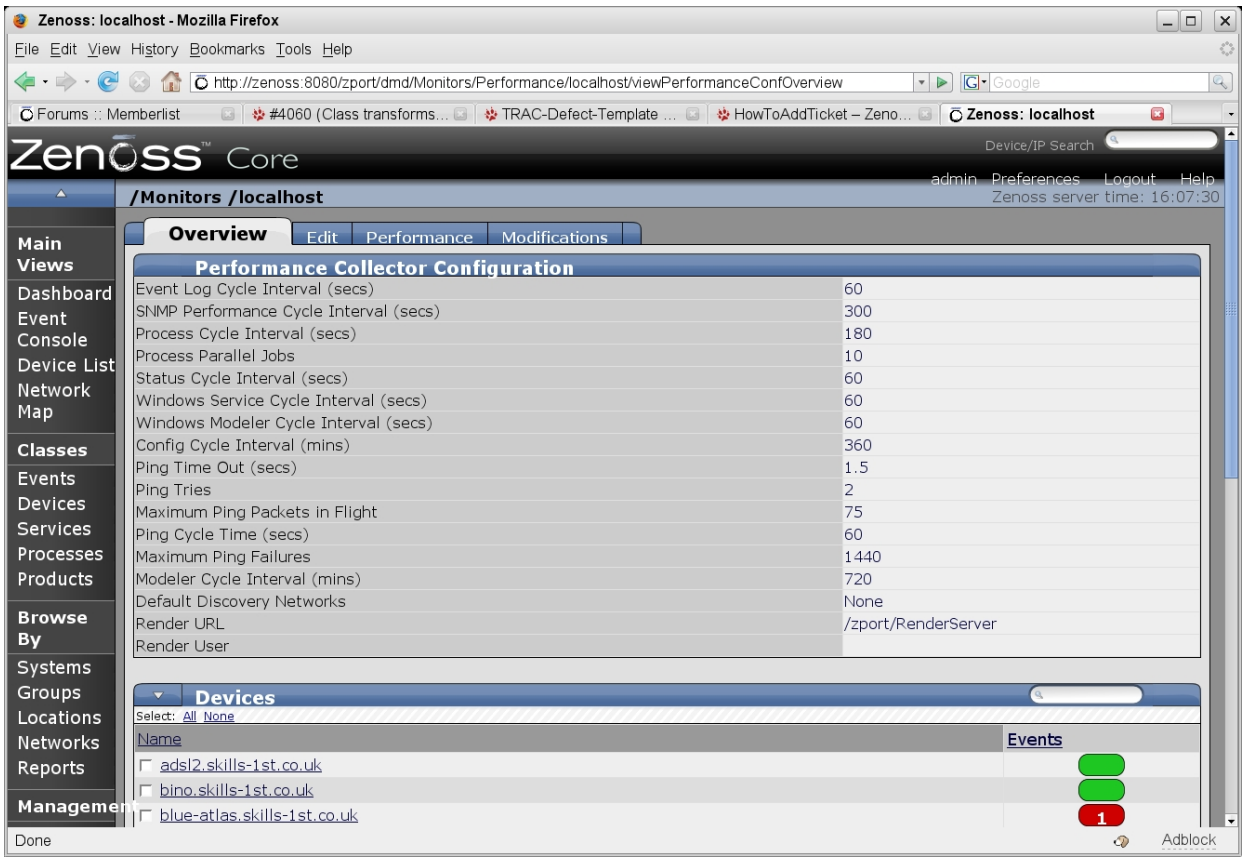

Figure 9: Default parameters for localhost Collectors

Parameters to note particularly are:

- SNMP polling cycle 300 secs (5 mins)
- Polling for processes 180 secs (3 mins)
- Status polling for TCP/UDP services 60 secs (1 min)
- Polling for Windows services 60 secs (1 min)
- Windows WMI poll 60 secs (1 min)
- Ping polling 60 secs (1 min)

3.1 zenping

The most basic level of availability checking is to ping-poll. The **zenping** daemon will, by default, ping-poll each interface, every minute. An interface down event is generated when the ping fails to get a response. This event is automatically cleared when a similar ping is successful; meantime, while an interface remains down, the count field of the event is increased.

The zenping daemon can detect when the network path to a device is broken, for example if a single-point-of-failure router is down. In this case, an event is generated

with an eventState field of *Suppressed* and the summary field reports not only the interface for which the ping failed, but also the causal device; for example:

ip 10.191.101.1 is down, failed at bino.skills-1st.co.uk

All other device availability monitoring is dependent on ping access. Once a ping has failed, SNMP, process, TCP/UDP service and windows service monitoring will all be suspended until ping access is restored. The count field of the higher level monitoring events will not increase until ping access is resumed.

Also note that if there is no ping access, no performance information will be collected. If a device really does not support ping, perhaps because of firewall restrictions, then ensure that the zProperty *zPingMonitorIgnore* is set to *True;* this will permit SNMP and ssh availability monitoring and performance data collection.

The logfile for zenping is *zenping.log* in *$ZENHOME/log*.

3.2 zenstatus

The **zenstatus** daemon can be configured to check for access to various TCP and/or UDP ports on both Windows and Unix architectures. By default, it checks every minute. Zenoss comes with a huge number of services pre-configured; these can be examined from the *Services -> IpService* left hand menu. By default, none of these service monitors are active. Service monitoring for a device can be configured from the device's main page – use the *OS* tab and the table dropdown menu beside *IP Services* to add a service to be monitored for this device.

As with ping polling, a “good news” service event for a device automatically clears a similar “bad news” event and the count field of the event increases whilst the service remains down.

The logfile for zenstatus is *zenstatus.log* in *$ZENHOME/log*.

3.3 zenwin

The **zenwin** daemon monitors Windows services (not TCP / UDP services). These can be examined from the *Services -> WinService* left hand menu. By default, none of these monitors are active. Windows service monitoring for a device can be configured from the device's main page – use the *OS* tab and the table dropdown menu beside *Win Services* to add a service to be monitored for this device.

zenwin uses the Windows Management Instrumentation (WMI) interface to access services on the remote system every minute, by default. The zProperties for a device (or device class) must be configured to allow access to WMI before windows service polling can be successful.

As with ping polling, a “good news” windows service event for a device automatically clears a similar “bad news” event and the count field increases on subsequent failed polls.

The logfile for zenwin is *zenwin.log* in *$ZENHOME/log*.

3.4 zenprocess

zenprocess monitors Windows and Unix systems for the presence of processes. In a Unix context, this would be whether the process appears in a *ps -ef* listing; in a Windows context, the process must appear in the Windows Task Manager (and note that this check **is** case sensitive on both architectures). Monitoring is every 3 minutes, by default.

Configuration of process monitoring for a device is similar as for services – use the device's main page -> *OS* tab and the table dropdown menu beside *OS Processes* to add a process to be monitored.

Process monitoring is actually achieved using the Host Resources Management Information Base (MIB) of SNMP, by retrieving the **hrSWRun** table. This means that if SNMP access to a device is broken, there will be no process information.

As with the other availability daemons, "good news" events clear "bad news" events and the count field increases on subsequent failed polls.

The logfile for zenprocess is *zenprocess.log* in *$ZENHOME/log*.

3.5 zenperfsnmp

zenperfsnmp polls each device every 5 minutes, by default. It can collect both SNMP performance information and status information for processes. **Process** monitoring is achieved using the SNMP Host Resources MIB so that if an SNMP agent fails, then process monitoring is also affected. This is **not** the case for Windows service monitoring. TCP / UDP service monitoring also does **not** rely on SNMP on any platform.

Within 5 minutes of an SNMP poll failure, an "snmp agent down" event should be generated. Within a further 3 minutes there should be an "Unable to read processes on device .." event, if process monitoring is configured. Note also that the count field for individual missing process events should stop increasing. Counts for missing service events will be unaffected by the loss of SNMP. While SNMP access to the device remains broken, the count field for the "Unable to read processes on device .." event will increase every 3 minutes.

3.6 Availability monitoring daemons and device status pages

Each device has its own Device Status page. There is an overall status icon button at the top of the page that fundamentally reports ping access (if ping access is enabled) – green for good status; red for bad. Note that if a device's zProperties are customised **not** to ping-monitor, then this overall status button will always remain green!

The top right-hand panel of the Device Status page reports **component** status for other monitored elements such as interfaces, services and processes. The *Other* category includes status for events reported via syslog or Windows event logs, where the colour of

the button represents the severity of the event. Note that Acknowledging such events has no effect on the Device Status page.

4 Syslog events

The Unix syslog mechanism is pervasive throughout all versions of Unix / Linux although slightly different versions and formats exist. There are also open source implementations of syslog for Windows systems and many networking devices also support the syslog concept.

Typically system messages are output to one or more log files such as */var/log/messages*. The syslog subsystem can also be configured to send syslog messages to a central syslog rather than holding files on each system. The well-known default port for forwarding syslog messages is **UDP/514**.

A standard syslog system is configured by the *syslog.conf* file, typically in */etc* . A newer version of syslog is implemented on some systems, **syslog-ng**, which has greater filtering capabilities. The syslog-ng configuration file is typically */etc/syslog-ng/syslog-ng.conf*.

A syslog message includes a **priority** and a **facility**. The priorities are:

0	emerg
1	alert
2	crit
3	err
4	warning
5	notice
6	info
7	debug

Facilities include:

auth (4)	authpriv (10)
cron (9)	daemon (3)
ftp (11)	kern (0)
lpr (6)	mail (2)
news (7)	syslog (5)
user (1)	uucp (8)

These definitions can be found in *syslog.h* (typically in */usr/include/sys*). Both priority and facility are encoded in a single 32-bit integer where the bottom 3 bits represent priority and the remaining 28 bits are used to represent facilities.

For example, if the facility/priority tag is <22>, this would be 00010110 in binary, where the bottom 110 represents a priority of 6 (info) and the top 00010 represents a facility of 2 = mail.

4.1 Configuring syslog.conf and syslog-ng.conf

Any device that is going to report syslog events to Zenoss must have its syslog.conf file configured with the destination address of the Zenoss system. The original syslog.conf permits filtering based on priority and facility so, a catch-all statement to send all events to the Zenoss system, would be:

```
*.debug        @<IP address of your Zenoss system>
```

syslog-ng.conf requires at least a **source**, a **destination** and a **log** statement. syslog-ng offers superior filtering over the original syslog so one or more **filter** statements may also be present.

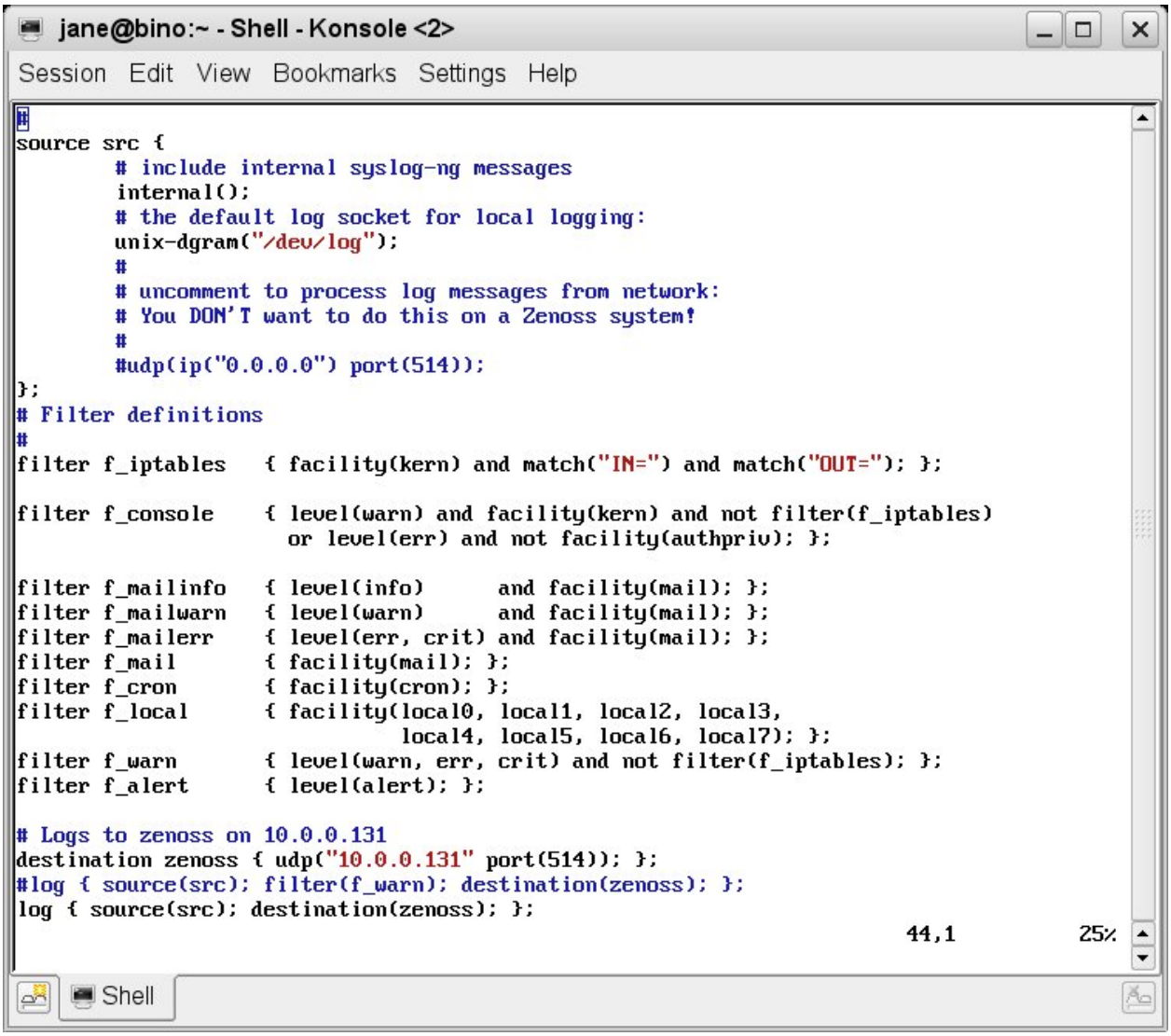

Figure 10: syslog-ng.conf to send all events to Zenoss system at 10.0.0.131 (no filtering active)

4.2 Zenoss processing of syslog messages

To collect syslog messages with Zenoss, the **zensyslog** process automatically starts on port UDP/514 and collects any syslog messages directed from other systems. zensyslog then parses these messages into Zenoss events. You must ensure that the syslog.conf file on the Zenoss system does **not** enable collecting remote syslogs or the syslogd and zensyslog processes will clash over who gets UDP/514 (it is possible to reconfigure either daemon, if required).

To examine the incoming syslog messages and the parsing that zensyslog performs, the level of zensyslog logging can be increased.

1. Use the *Settings* menu on the Zenoss left-hand menu and choose the *Daemons* tab.
2. Click the *edit config* link for the zensyslog daemon.
3. Change the following parameters and click *Save*:

   logorig select this

   logseverity *Debug*
4. Inspect the underlying configuration file in *$ZENHOME/etc/zensyslog.conf*.
5. The **logorig** line says to log the original incoming syslog message; it will be in *$ZENHOME/log/origsyslog.log*. Note that this parameter is unique to zensyslog and is useful for debugging.
6. The **logseverity** line is a generic Zenoss daemon parameter; a value of 10 is the maximum *Debug* level.
7. Don't forget to Save this change
8. Use the *Restart* link to recycle zensyslog. Alternatively, as the *zenoss* user, issue the command:

   ```
   zensyslog restart
   ```
9. Examine the zensyslog log file in *$ZENHOME/log/zensyslog.log*
10. A new incoming event starts with a line showing hostname and ip address, eg.

    ```
    host=zen241.class.example.org, ip=172.16.222.241
    ```
11. The next 2 lines show the raw message and the decoding for facility and priority.
12. Lines starting with *tag* show the zensyslog parsing process as it tests the incoming line against various Python regular expressions, hopefully ending with a *tag match* line.
13. If a match is successful, an eventClassKey may be determined
14. The last line for a parsed event should be a *Queueing event* .

Whenever different native event log systems are integrated there is almost inevitably a mismatch of severities. The following table demonstrates this.

Zenoss	**syslog priority**	**Windows**
Critical (red) (5)	emerg (0)	Error (1)
Error (orange) (4)	alert (1)	Warning (2)
Warning (yellow) (3)	crit (2)	Informational (4)
Info (blue) (2)	err (3)	Security audit success (8)
Debug (grey) (1)	warning (4)	Security audit failure (16)
Clear (green) (0)	notice (5)	
	info (6)	
	debug (7)	

Table 4.1.: Event severities for Zenoss, syslog and Windows

Note that the numeric value of Zenoss event severity **decreases** as events get less critical but that the priority of syslog events **increases** as events get less critical.

Default mapping from syslog priority to Zenoss event severity, is performed by */usr/local/zenoss/zenoss/Products/ZenEvents/SyslogProcessing.py* – search for *defaultSeverityMap* around line 163. The result is that:

- syslog priority < 3 (emerg, alert, crit) map to Zenoss severity 5 (Critical)
- syslog priority 3 (err) maps to Zenoss severity 4 (Error)
- syslog priority 4 (warning) maps to Zenoss severity 3 (Warning)
- syslog priority 5 or 6 (notice , info) map to Zenoss severity 2 (Info)

Out-of-the-box, all syslog events map to the Zenoss event class of **/Unknown** . *SyslogProcessing.py* in *$ZENHOME/Products/ZenEvents* is the code that parses any incoming syslog message and generates a Zenoss event.

The first section has a series of Python regular expressions to match against the incoming syslog line. Each expression is checked in turn until a match is found. If no match is found then an entry goes to *$ZENHOME/log/zensyslog.log* with *parseTag failed* .

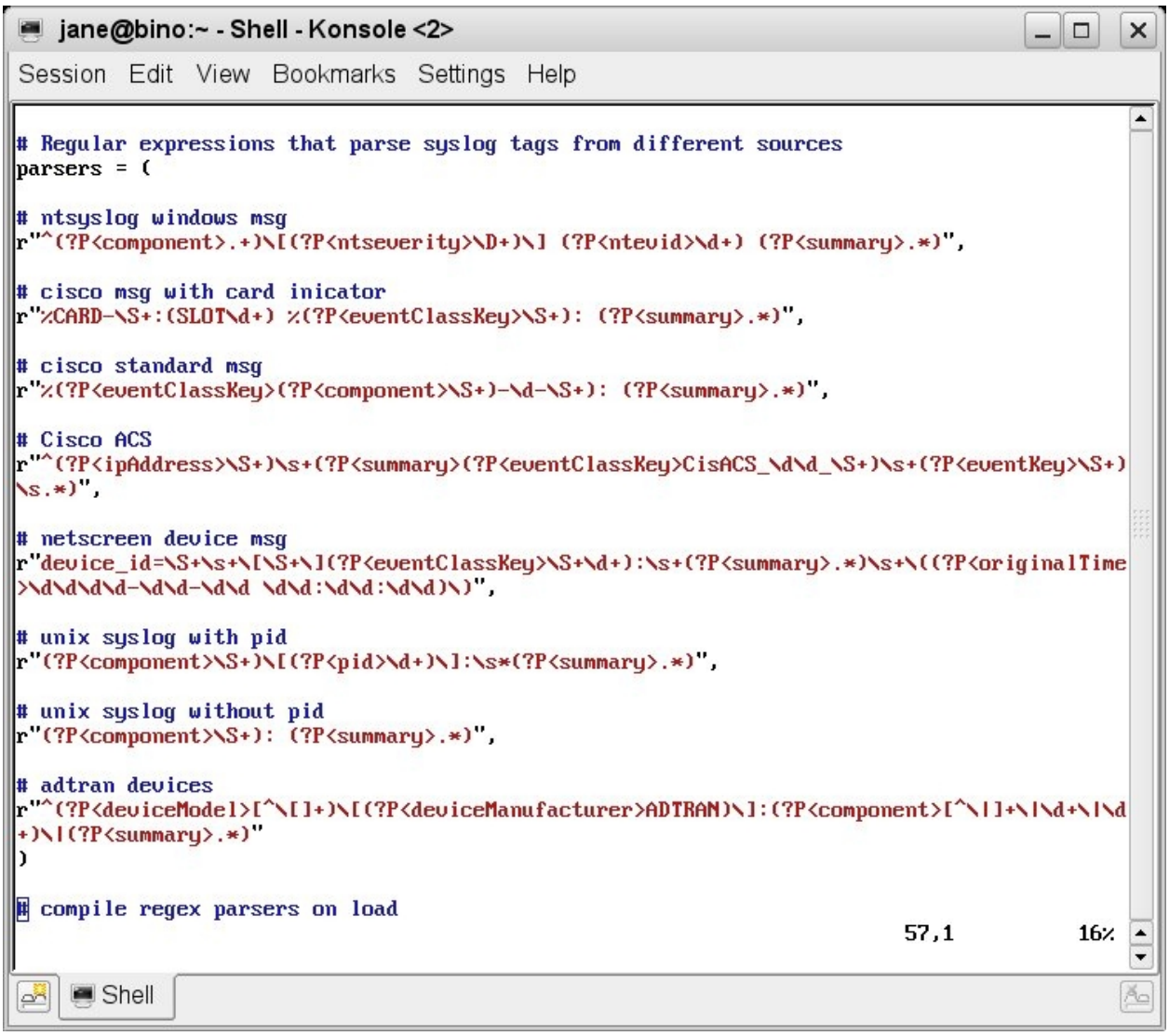

Figure 11: SyslogProcessing.py regular expressions to match syslog tags

The main body of SyslogProcessing.py starts by assigning values from the incoming event to Zenoss event class fields, as follows:

```
    def process(self, msg, ipaddr, host, rtime):
        evt = dict(device=host,
                   ipAddress=ipaddr,
                   firstTime=rtime,
                   lastTime=rtime,
                   eventGroup='syslog')
```

At this stage, no account of duplicates is taken so the firstTime and lastTime fields are both set to the timestamp on the incoming event. Note that the Zenoss eventGroup field is hardcoded at this stage to *syslog* .

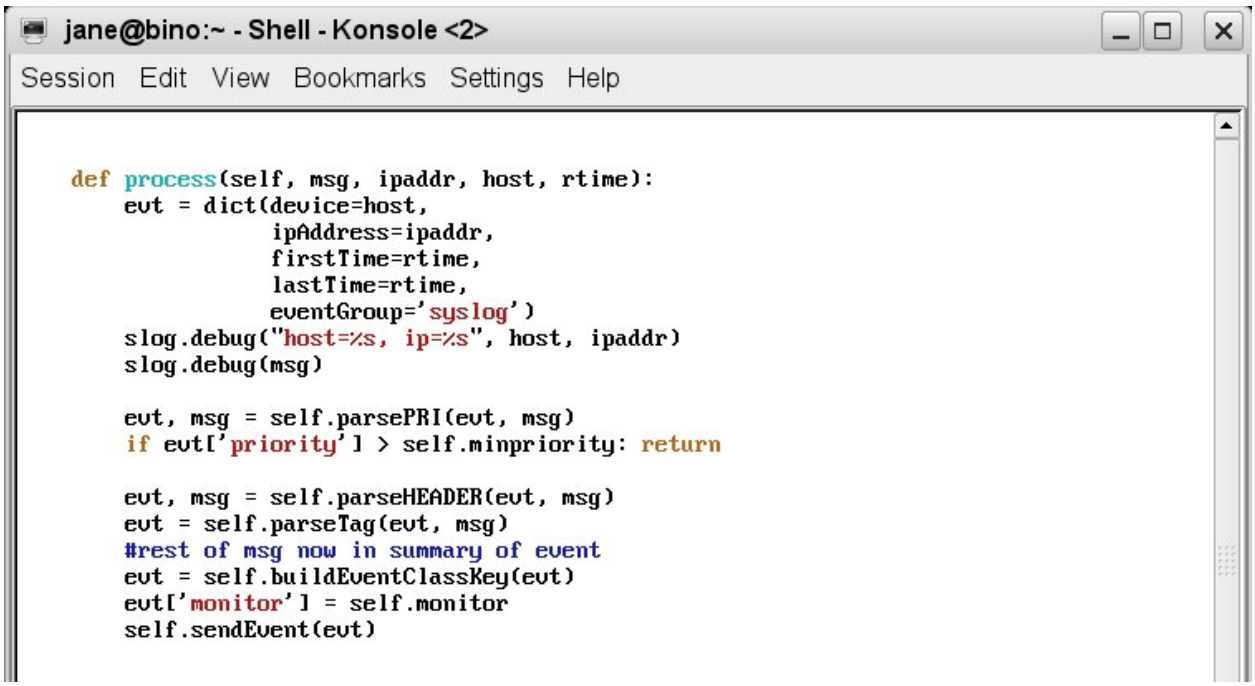

Figure 12: SyslogProcessing.py process main routine

parsePRI is the Python function called to parse out the syslog priority and facility.

The *defaultSeverityMap* function is called from within the parsePRI function to set the severity field of the Zenoss event.

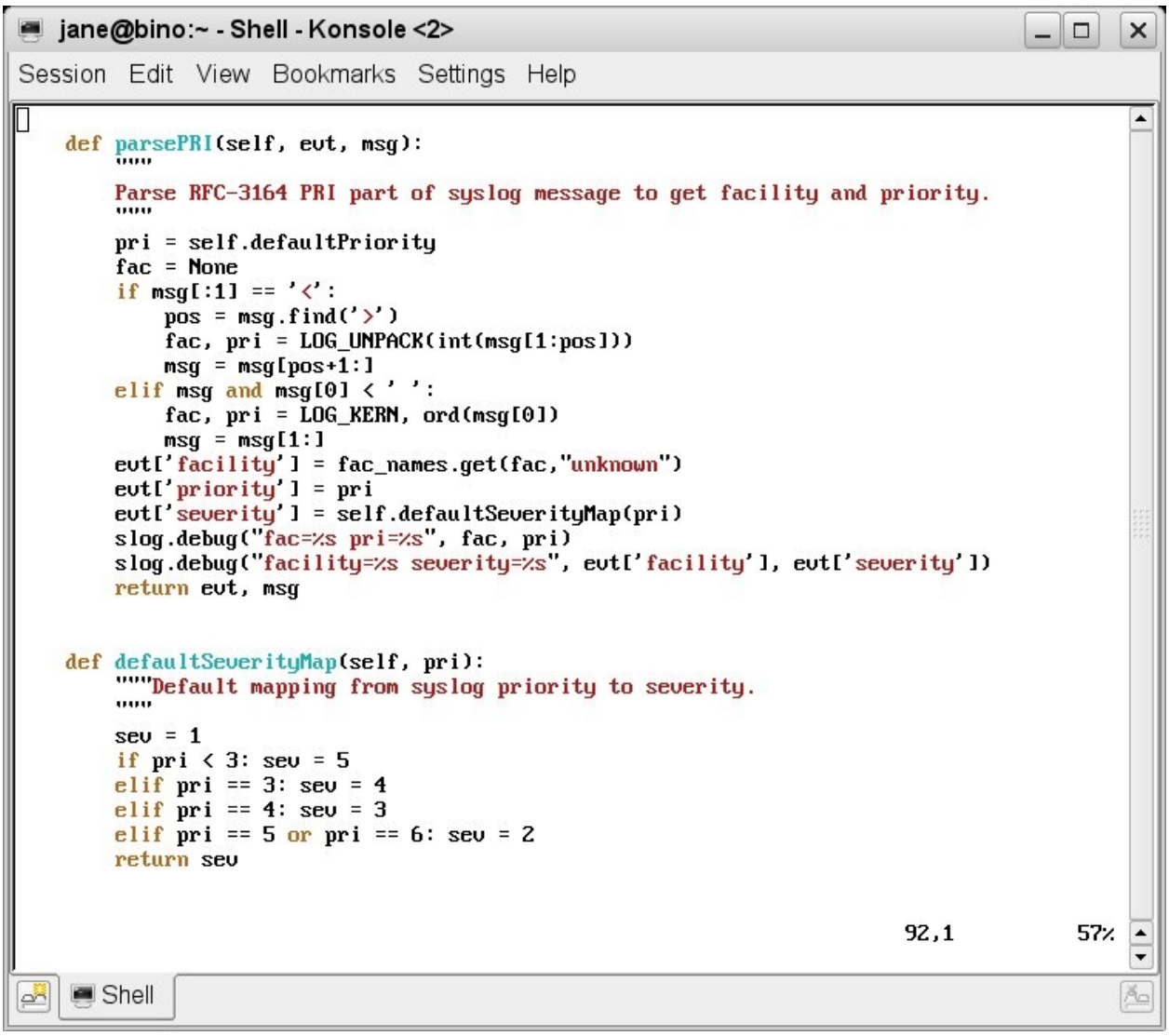

Figure 13: SyslogProcessing.py parsing of priority, facility and severity

Next, the *parseHEADER* function is called to extract the timestamp and host name from the incoming event. If the hostname does not exist then an attempt is made to lookup the name from the IP address using a gethostbyname call. The device field of the Zenoss event is set at the end of this function.

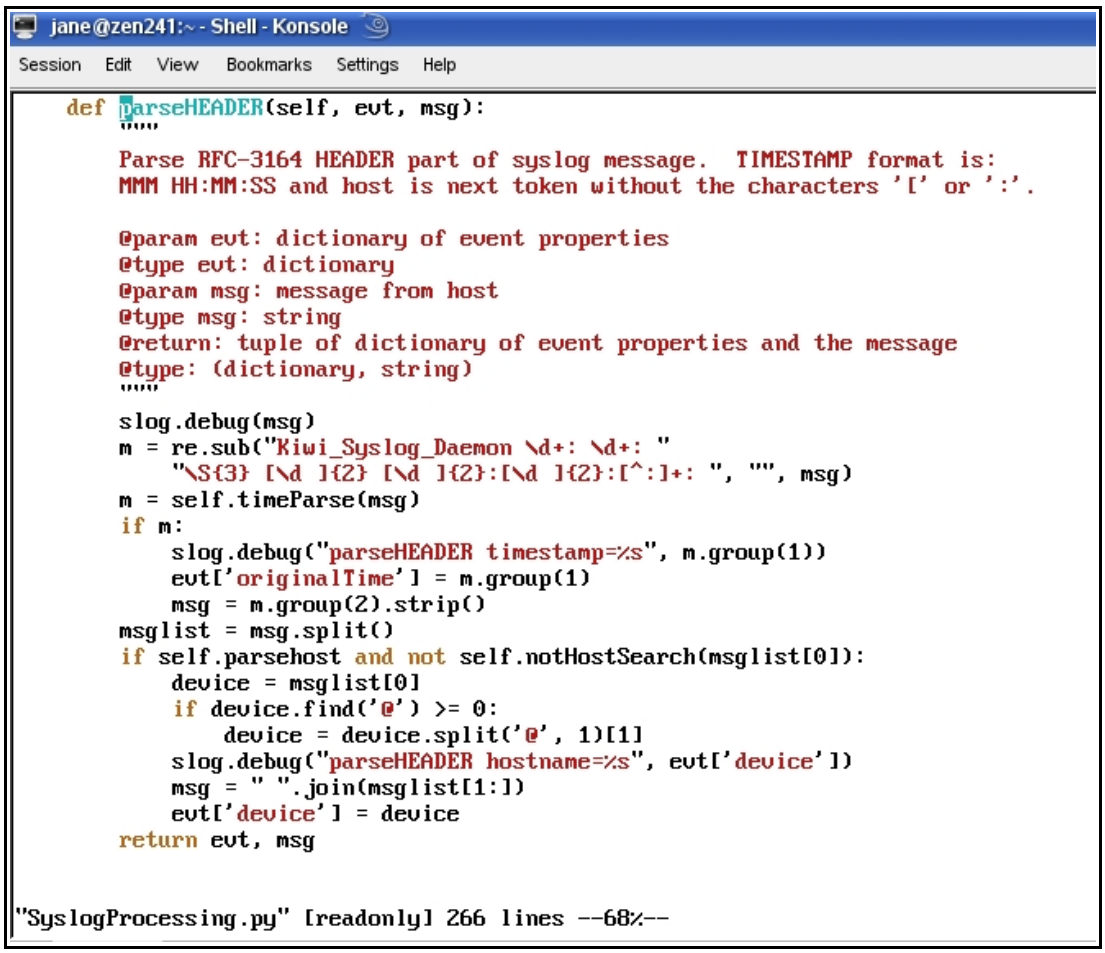

Figure 14: SyslogProcessing.py processing the header information

The *parseTag* function is called to parse out the syslog tag, using the regex expressions at the beginning of the file. If no match exists then a *parseTag failed* message is logged. The end of the function returns the remainder of the incoming message in the Zenoss event summary field.

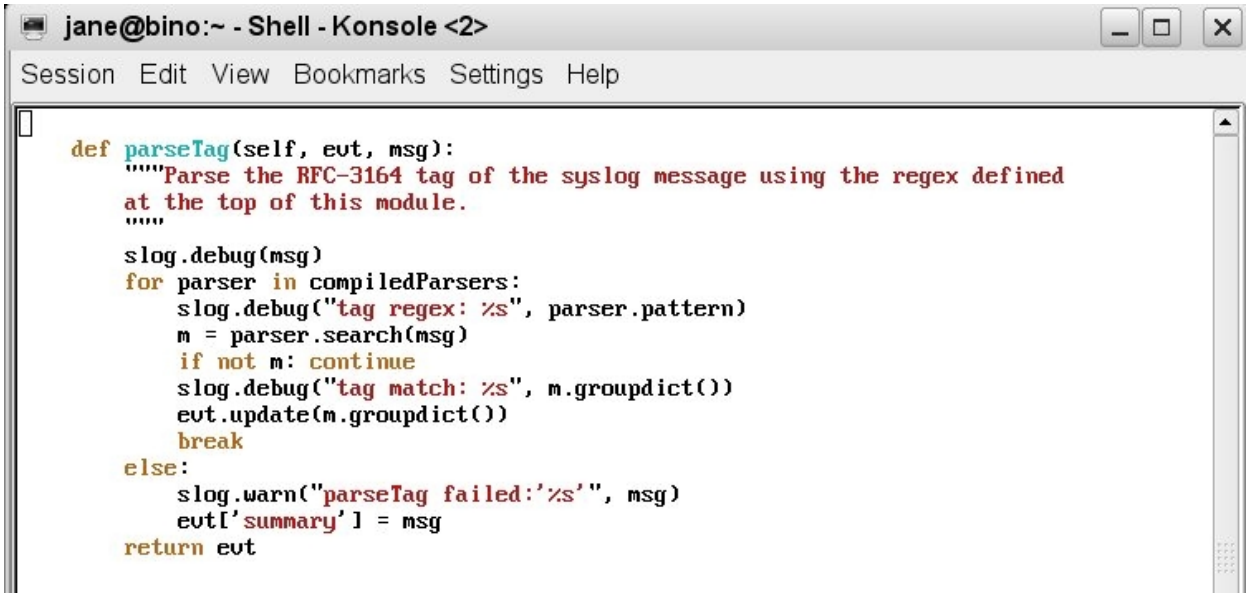

Figure 15: SyslogProcessing.py parsing the syslog tag

The crux of event processing in Zenoss is to derive an **eventClassKey** – this is done with the *buildEventClassKey* function.

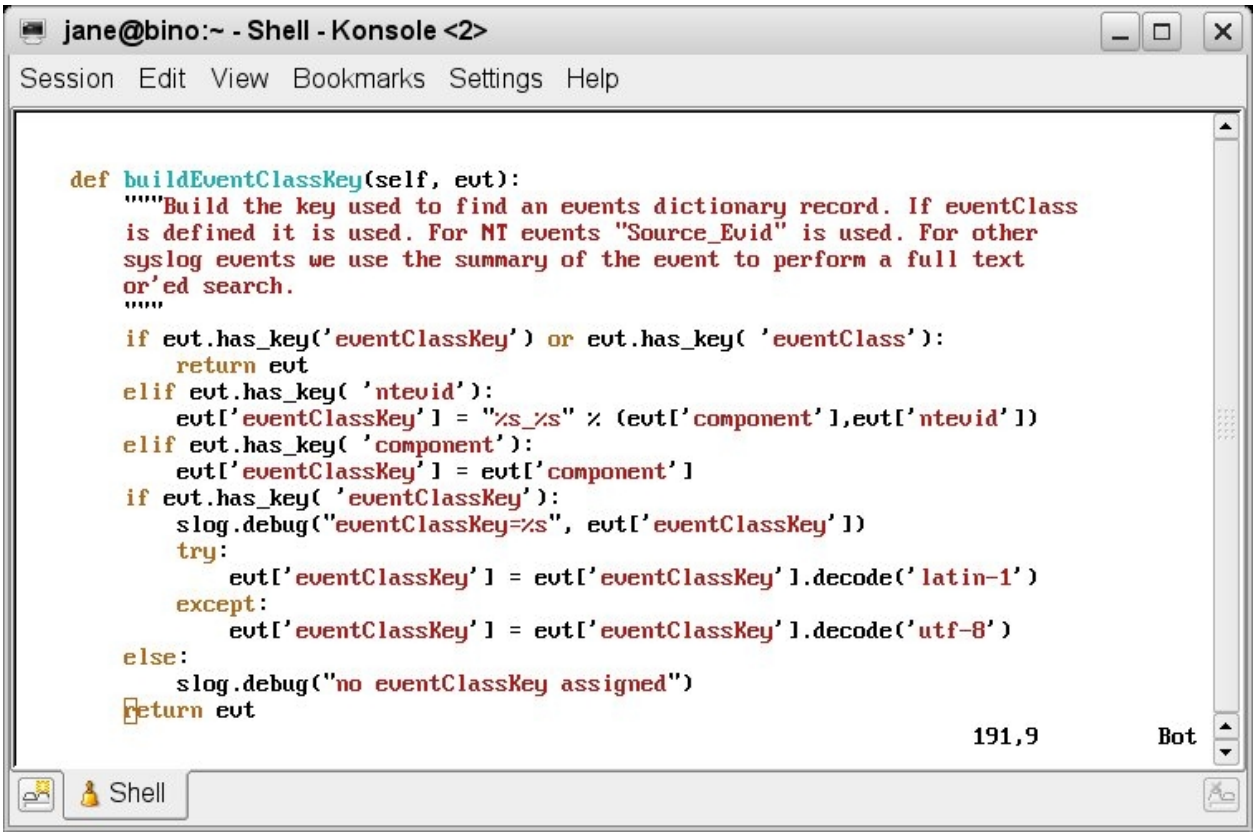

Figure 16: SyslogProcessing.py determining the EventClassKey

Note that if the event has the **component** field populated then that is used as the eventClassKey after checking for a pre-existing **eventClassKey** and for an **ntevid** field.

5 Zenoss processing of Windows event logs

The **zeneventlog** daemon is responsible for processing events from Windows event logs . It uses the WMI mechanism so, like the **zenwin** daemon for monitoring Windows services, the Windows zProperties for a device (or device class) must be configured correctly. From the dropdown table menu of a device's status page, choose *More -> zProperties*. Scroll down to the *zWin...* properties and check you have the following settings. Don't forget to *Save* if necessary.

- zWinEventlog True
- zWinEventlogMinSeverity 16 (this will gather ALL events)
- zWinPassword < the correct password for Administrator>
- zWinUser Administrator
- zWmiMonitorIgnore False

Note especially the **zWinEventLog** property to turn on/off event log collection and **zWinEventLogMinSeverity** that provides a crude filtering mechanism (see the earlier Table 4.1 on page 28 for the different severities for Windows Event logs).

It is the **zeneventlog** daemon in Zenoss that receives incoming Windows events and parses them into Zenoss events. Typically, the Source field on the Windows event maps to the component field in the Zenoss event; the Zenoss eventClassKey is composed of the Windows <*Source*>_<*Event ID*> (eg. Perflib_2003); the Zenoss eventGroup becomes the Windows log file name (Application, Security, etc) and the Windows Event ID is mapped to the Zenoss ntevid field.

Many Windows event log events are automatically mapped to event classes but they may have a low severity (such as *Debug*) and they may have their zEventAction event zProperty set to *history* so that they do not appear in the status table of the events database.

Watch out for events of class */Status/Wmi/Conn*, typically after a Windows system reboot. Until this event is moved to history, no other events will be received from the WMI interface on the Windows system as the zenwin daemon will not reconnect and zeneventlog will receive no events.

There is also a syslog utility available for Windows systems from Datagram Consulting at http://syslogserver.com . The client utility is SyslogAgent and is made available under the GNU license. Syslog server utilities for Windows are also available as

chargeable products. This means that Windows event logs can also be collected with the zensyslog daemon.

Note that the Syslog agent is capable of being configured to monitor Windows application log files, in addition to the standard Windows event logs. When monitoring the standard event logs, there are better filtering capabilities than when using zeneventlog.

6 Event Mapping

Zenoss events are categorised into a hierarchy of event **Classes**, many of which are defined out-of-the-box but which can easily be modified or augmented. The process of **Event Class Mapping** is about associating an incoming event with a particular Zenoss Event Class (setting its *eventClass* field) and, potentially, modifying other fields of that event by using an event **transform**.

Event classes and subclasses are treated identically from the point-of-view of event class mapping. The class hierarchy can be useful in that event **context,** as implemented by event zProperties (such as zEventSeverity, zEventAction), follows the normal rules for object inheritance – if zEventAction is set to *drop* on the event class */Ignore* , then any subclasses of */Ignore* will also inherit that property.

Notable out-of-the-box event zProperties are that /Ignore classes and subclasses drop incoming events (ie. they do not appear in either the status or the history databases); /Archive classes and subclasses automatically move events to the history database.

Most event classes have one or more mappings associated with them – these are known as **instances** . Note that an event does **not have** to have any mappings associated, in which case an event of that class will only appear in an Event Console if the daemon that generates the event, assigns the event class at that time (*/Perf* events may well come into this category, for example). Out-of-the-box event class mappings are defined in *$ZENHOME/Products/ZenModel/data/events.xml* . They can be inspected from the Zenoss GUI by selecting *Events* in the left hand menu and drilling down the subclass hierarchy under the *Classes* tab. Alternatively, the *Mappings* tab shows a more mapping-centric list, rather than a class-centric list.

Most out-of-the-box event class mappings simply match on the **eventClassKey** field which is populated by the native event parsing mechanism (such as zensyslog, zeneventlog, zentrap). These mechanisms may generate several different events with the same eventClassKey field; thus other techniques are needed to distinguish between such events and potentially to separate them into different event classes.

The sequence number in an event mapping gives the order in which mappings are tested against the incoming event. Depending on which mapping actually matches (if any) will determine the resulting eventClass of the event.

6.1 Working with event classes and event mappings

Events are organised in an object-oriented hierarchy; thus attributes assigned to a "parent" event class are inherited by a "child" event subclass.

New event classes can be defined by selecting the left-hand *Events* menu, drilling down to a relevant subclass, if required, and then using the dropdown table menu alongside *SubClasses* to *Add New Organizer*. The name supplied is the name of the new event class. For example, drill down to the */Security* event class and create a new subclass called *Su*.

The simplest way to map an event to a new or different class is to start from an existing event in the Event Console. The following scenario explains this, creating a new event class mapping called *su* which maps an incoming event to the event class */Security/Su*.

1. Generate a syslog FAILED SU event at the Zenoss system.
2. Open an Event Console that shows the event and inspect its details, including the *Details* tab.
3. Select the event and using the table menu, select *Map Events to Class* . Select your new */Security/Su* class from the dropdown list. You should be shown the event class mapping panel. Click the *Edit* tab.
4. You should find that the name of the new event class mapping is set to s*u* and the Event Class Key is set to s*u* (note lower case s in both cases). The *eventClassKey* field is actually derived from the *component* field of the incoming event in SyslogProcessing.py (around line 256). The summary field of the event should have been copied into the mapping Example box.
5. Add a text string to the Explanation box such as "Auto added by event mapping".
6. Add a text string to the Resolution box such as "This is a dummy resolution".
7. Open a Zenoss GUI window that shows all Su events (you may find it useful to have several browser tabs open to focus on different aspects of the Zenoss GUI). Select all the Su events and *Move to History*.
8. Generate a new Su event.
9. Check the details of the new event in the Event Console. The event should have mapped to eventClass */Security/Su* . The severity should be *Info* (blue). The details of the event should show the eventClassMapping field set to */Security/Su/su* .

Any existing event mapping can be modified starting from the *Events* left-hand menu and use the *Mappings* tab. The *Show All* button at the bottom of the first page, will display all mappings in alphabetical order. Once the mapping is selected, changes can be made from the *Edit* tab.

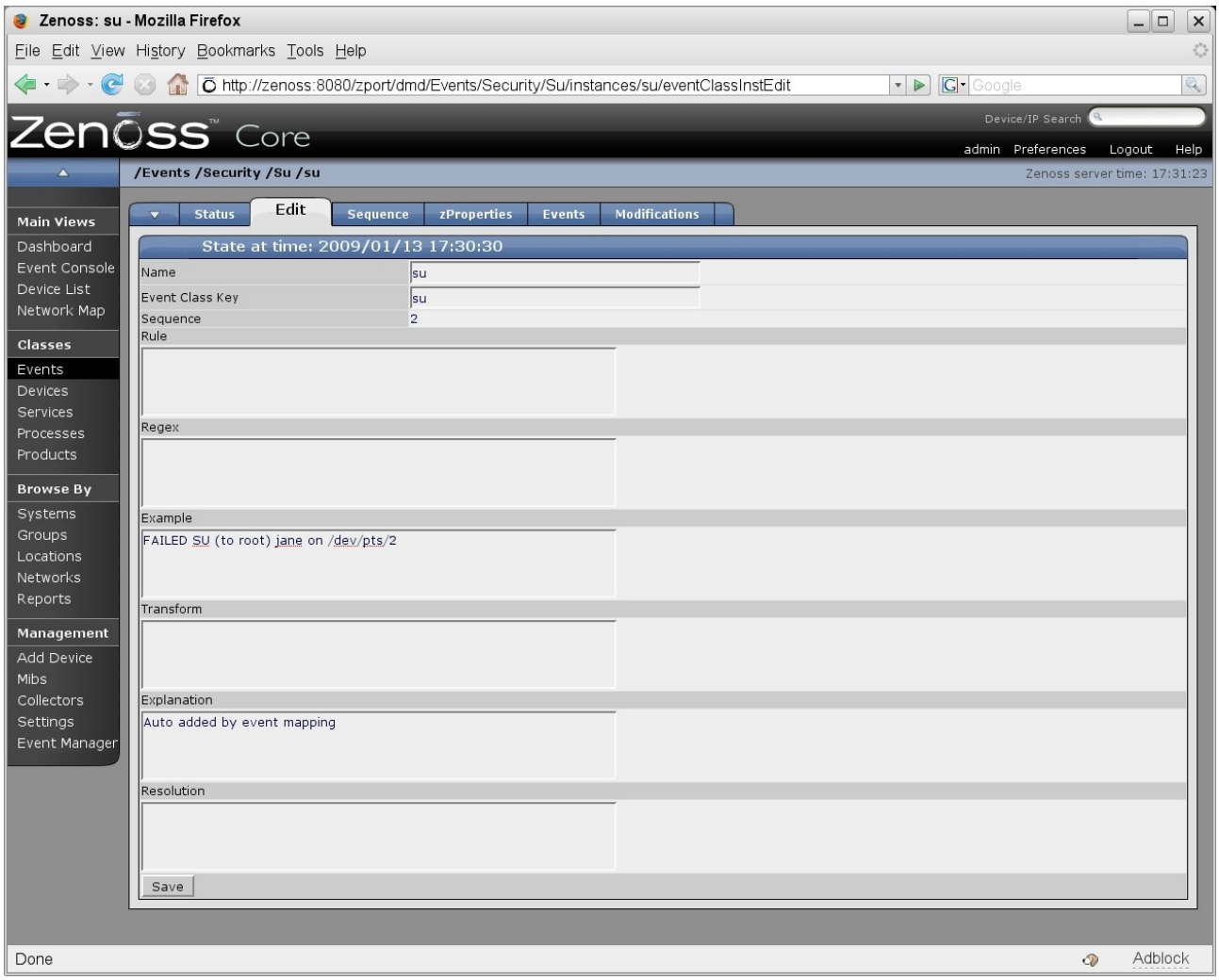

Figure 17: Edit dialogue for event class mapping

Whenever you change an event mapping, it is advisable to clear (*Move to History*) any existing events of that category before testing the new configuration. This is achieved by selecting one or more events and using the dropdown table menu.

When you are working with event mappings, don't forget the *Event* tab which filters an Event Console by Event Class; also, any Event Console has a filter available at the top right of the screen.

It is useful to refer to event classes using the **breadcrumb** path seen at the top of a page, such as */Events/Security/Su* .

Test events can be created from the Event Console dropdown table menu, *Add Event* Note that this is **only** available from an Event Console reached by starting from the left-hand *Events* menu, **not** from a generic console or the events for a specific device (this changed between Zenoss 2.3 and 2.4).

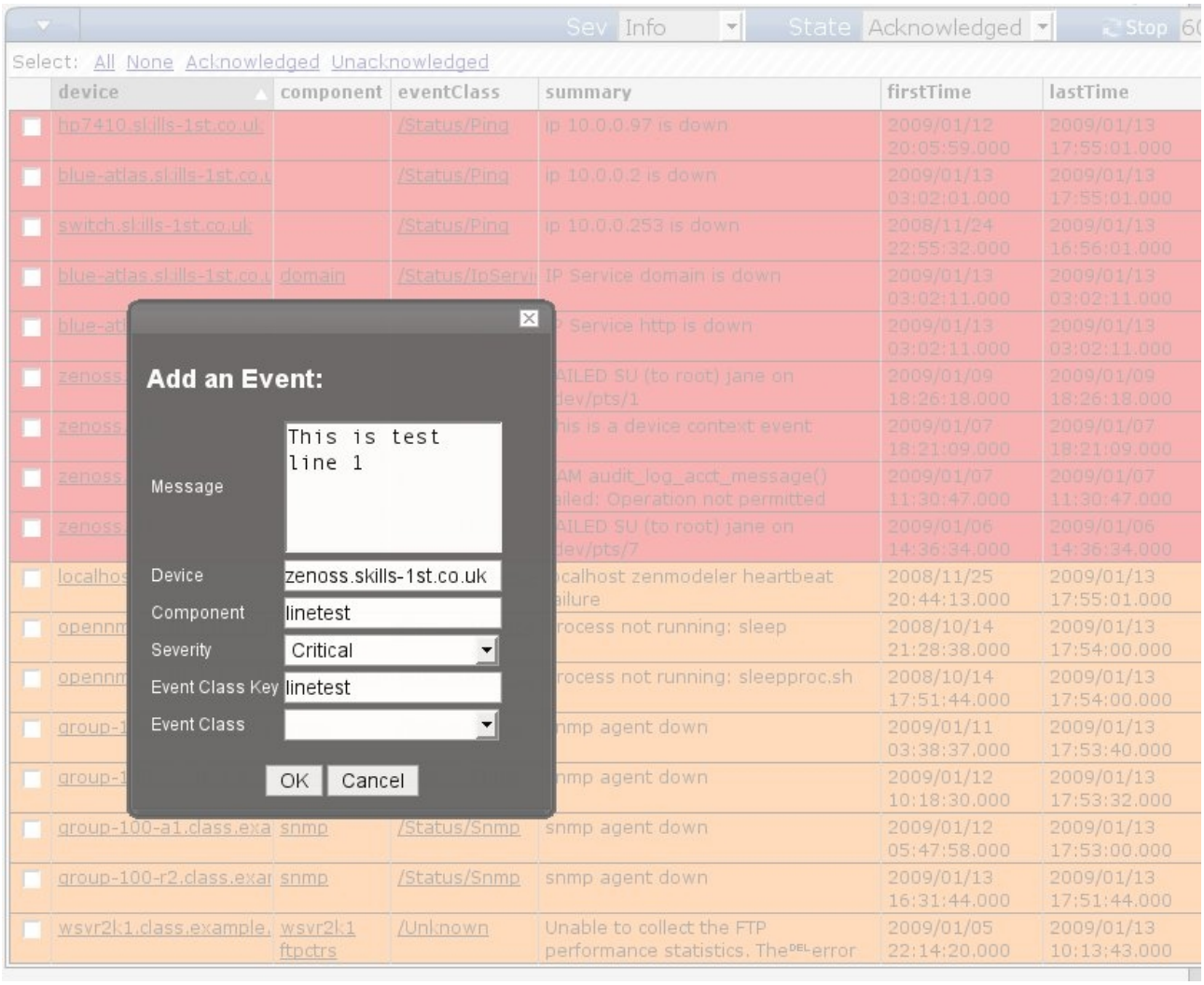

Figure 18: Dialogue to create a test event

Alternatively, the command line *zensendevent* can be used (you should ensure you are the zenoss user). This takes parameters:

- -d device
- -p component
- -k eventClassKey
- -s severity
- -c eventClass
- --port=PORT default is 8081
- --server=SERVER default is localhost
- --auth=AUTH default is admin:zenoss
- The remainder of the line after these options is used for the summary field (strictly the Message field in the GUI dialogue populates the event *summary* field)

6.2 Rules in event mappings

The **Rule** element of an event class mapping uses Python expressions to test any instantiated field of the incoming event against a value. Expressions can be complex including Python method calls and logical ANDs and ORs. The default event fields that are defined, are given in Appendix D3 of the Zenoss Administration Guide. **Note** that some of these fields are **not** actually available at event mapping time – notably **evid, stateChange, count, dedupid, suppid** and **eventClassMapping** .

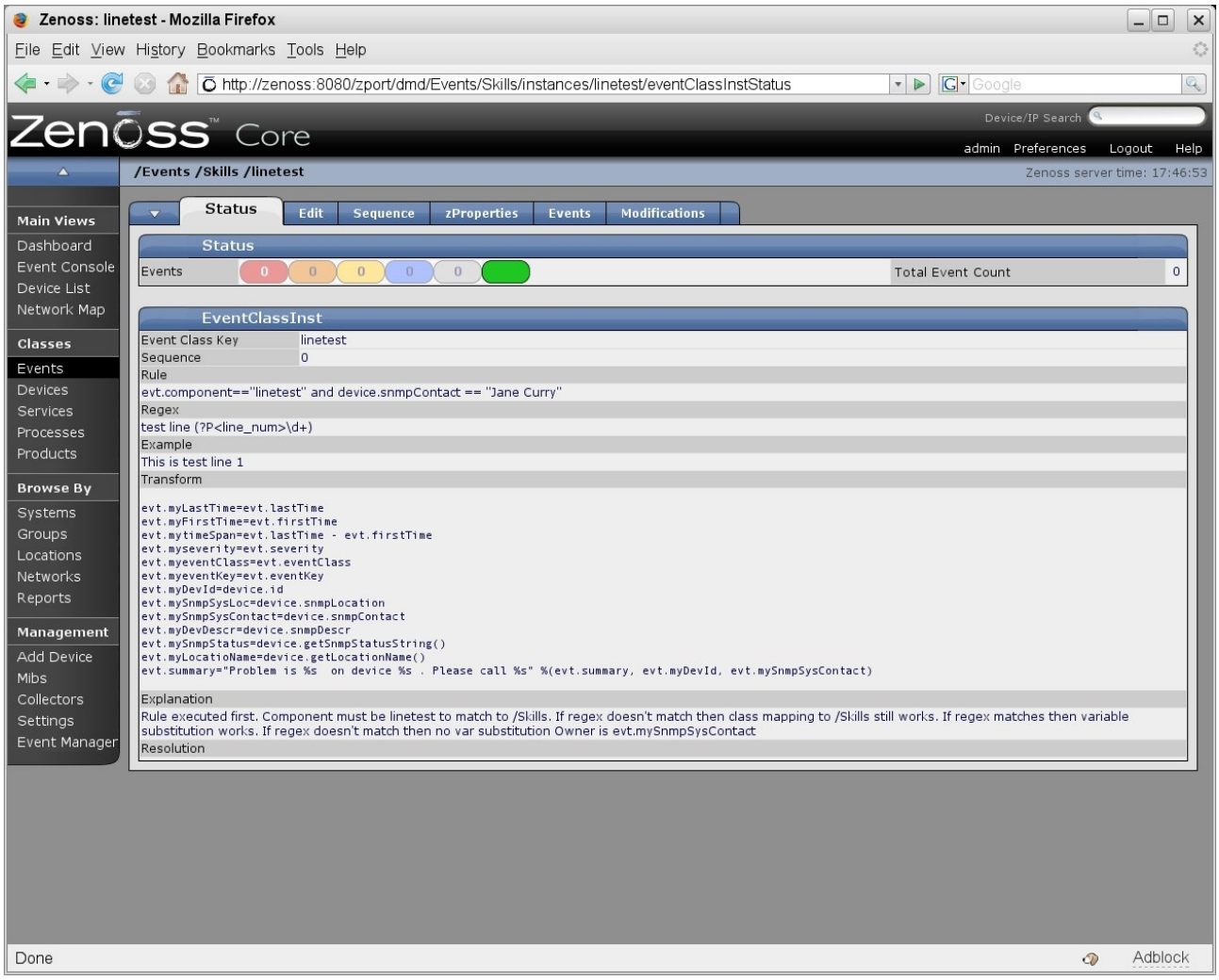

Figure 19: Event mapping linetest, showing complex Rule testing event and device attributes

The Rule element can also use Python expressions to test for values of attributes of the **device** that generated the event. Some of the methods and attributes that are available for devices are documented in Appendix D2 of the Zenoss Administration Guide, under the section on TALES expressions (**T**emplate **A**ttribute **L**anguage **E**xpression **S**yntax is part of Zope. Zope is the application server that Zenoss is built on).

The Rule element will **only** be used if the *eventClassKey* field in the mapping has achieved a match with the incoming event. After that, if a Rule exists, it must be satisfied before this mapping (and hence class) is applied.

6.3 Regex in event mappings

The **Regex** element of an event class mapping can be used to parse the summary field of the incoming event, which is presented by the parsing daemon (zensyslog, zeneventlog, zentrap). The Regex element uses the Python format for regular expressions and can use the Python **named group** syntax to not only check for literal strings but also to define regular expressions for variable parts of a string, and associate that variable part with a name. Variable parts of the string are captured into Python **named groups** – this means that:

- You can have one expression match lots of similar but different incoming events
- The variable part (typically between the *(?P and \S+*)) can be passed to the rest of the event processing mechanism as a named field of the event.
- Thus, in the Regex
    - exit before auth \(user '(?P<eventKey>\S+)', (?P<failures>\S+) fails \): Max auth tries reached
    - (?P<eventKey>\S+) will parse the characters after *user ′* upto the next single quote and place that string into the eventKey field of the event. Similarly (?P<failures>\S+) will hold the string that follows a comma and space and is ended by space and *fails*.
    - Matching the literal string representing a bracket requires the backslash escape or the bracket will be interpreted as a metacharacter.
    - The rest of the event summary must match the literal text in the Regex; however, other text can appear beyond the end after *tries reached* .
    - The Example box usually shows a sample event summary that is matched by the regular expression in the Regex box.

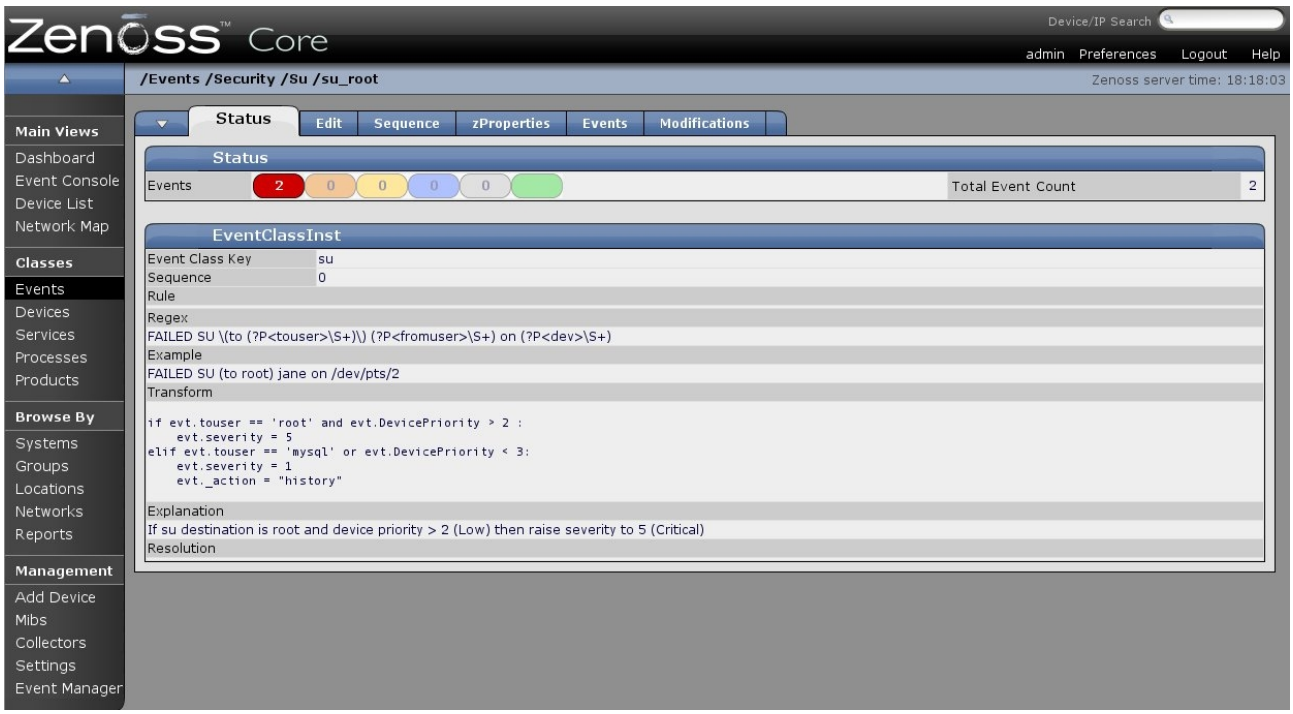

Figure 20: Event mapping dialogue with Regex for su failure

Hence, the event summary field can be parsed to generate new, user-defined fields for the event which will be shown in the Details tab of an event's details and can be used in any subsequent event transforms.

The Regex element is **only** used if both the eventClassKey and the Rule (if any) are satisfied. If the Rule fails, the Regex will not be tested, nor will any named group, user-defined fields be generated. If a Rule does **not** exist and the Regex does not match, the user-defined fields will not be generated and the event class mapping to this event class will fail. No event transforms will take place. If a Rule **does** exist and is satisfied but the Regex fails then any user-defined fields will **not** be generated but the event class mapping **will** be successful and any mapping transform **will** take place.

6.4 Other elements of event mappings

The **Example** element of an event class mapping is a sample string that is useful when constructing a Regex. The Regex will turn red if the Regex does not match the Example string when the *Save* button is used.

The **Explanation** and **Resolution** elements of an event class mapping are strings that can be configured to provide further information to Zenoss users. They appear in the *Details* tab of an event's detail. **Note** that these elements can only be literal strings; they cannot use either standard or user-defined fields from the event.

The combination of eventClassKey, Rule and Regex determine the event class that will be associated with the incoming event and what transforms (if any) will take place. There may still be multiple combinations of these that satisfy any given incoming event.

If so, the *Sequence* tab is used to decide the precedence of evaluation of matching event mappings. The mappings will be tested from the lowest to the highest sequence number. Once a match is found, any subsequent mappings (with higher sequence numbers) will be ignored. Generally, a mapping with more specific matching criteria will have a lower sequence number.

A particular example of event mappings that use sequence numbers, is the event class mapping called **defaultmapping** which must have an eventClassKey of *defaultmapping* . There are at least 6 mappings, all called *defaultmapping* , out-of-the-box. Each maps to a different class. A default mapping is a special case that is used by the event mapping process if no match can be found for the eventClassKey field (note that if the eventClassKey field does not exist then no mapping at all will be applied). In the case where an eventClassKey match is not found, the mapping process re-evaluates looking for a match with the special eventClassKey of *defaultmapping* . It is possible to create new mappings, either with the name of *defaultmapping* or, indeed, with a different name, provided the eventClassKey is *defaultmapping* . The sequence numbers of all such default mappings should be adjusted to prioritise these default mappings.

7 Event transforms

Transforms can be used to do a number of clever things! Some default event fields can be modified; new, user-defined fields can be created; fields can be retrieved from events already in the MySQL database. You can have simple assignments of field values or set them based on complex Python programs. The transform mechanism can be applied in two ways:

- event class transforms
- event class mapping transforms

Prior to Zenoss 2.4, an event **class** transform was only used for events inserted directly to that exact event class by the parsing mechanism (zensyslog, zentrap, zensendevent, AddEvent with Event Class specified, etc). If a transform existed in an event class **mapping** that was used, the event class transform was **not** used.

Zenoss 2.4 introduced **cascading event transforms**. This changes things in two ways. Given an event class */Toptest* with a subclass of */T1***,** if an event arrives that already has class */Toptest/T1*, then the Toptest transform will be applied, followed by the T1 transform. If an event arrives that does not have a pre-allocated class but whose event class is determined to be */Toptest/T1*, by the Rule / Regex of the event class mapping, t1, then transforms will be applied in the order:

- Toptest class -> T1 class -> t1 event class mapping

It is perfectly possible for a transform to use user-defined event fields instantiated by earlier transforms; however, be very aware that if any statement in a transform fails (perhaps because a field doesn't exist), then the processing of that transform will stop at

that point and no further statements will be executed. Any further transforms **will** be executed (at least until an error is reached).

All transforms are executed once the Rule and Regex elements of a mapping have been successfully tested and after device and event context have been applied. Thus, at transform time, most of the standard event fields are available, **except** those populated at database insertions time (evid, stateChange, dedupid, count, eventClassMapping – and the firstTime and lastTime fields will be the same). Any user-defined fields created by the Regex are also available.

Event class transforms can be useful on the */Unknown* class to selectively change the class for events that would otherwise be */Unknown* .

Other than that, the two applications of transforms are very similar – it is basically Python code to modify event fields.

Note that if a transform tries to reference a field of an event that does not yet exist (like *count*) then that line of the transform **and any subsequent lines** will be ignored. Such an error will not trigger any error messages in the transform dialogue. Inspect the end of *$ZENHOME/log/zenhub.log* and *$ZENHOME/log/event.log* to see the error message reporting the absence of the attribute.

A class transform is configured starting from the E*vents* left-hand menu, by drilling down to the class in question (**not** the class mapping) and then use the top dropdown table menu to bring up the *More* -> *Transform* menu.

A mapping transform is specified as part of the same event mapping dialogue that defines the Rule and Regex fields. In each case, if the Python syntax is incorrect, when you use the *Save* button, then the transform is all displayed in red text, indicating an error.

Figure 19 on page 39 showed an event mapping called linetest which includes a transform to create several user-defined event fields, some based on values from the event and some with values from the device that generated the event. The event summary field is set to a string constructed from literal text, standard event fields and user-defined fields.

Transform

```
evt.myLastTime=evt.lastTime
evt.myFirstTime=evt.firstTime
evt.mytimeSpan=evt.lastTime - evt.firstTime
evt.myseverity=evt.severity
evt.myeventClass=evt.eventClass
evt.myeventKey=evt.eventKey
evt.myDevId=device.id
evt.mySnmpSysLoc=device.snmpLocation
evt.mySnmpSysContact=device.snmpContact
evt.myDevDescr=device.snmpDescr
evt.mySnmpStatus=device.getSnmpStatusString()
evt.myLocatioName=device.getLocationName()
evt.summary="Problem is %s  on device %s . Please call %s" %(evt.summary, evt.myDevId, evt.mySnmpSysContact)
```

Figure 21: Transform for the linetest event mapping

The linetest mapping transform also demonstrates that, at transform time, the firstTime and lastTime fields are the same; the value of *evt.mytimeSpan* is always 0.

Most of the user-defined fields are assigned to simple **attributes** of either the event or the device; for example, *evt.firstTime* , *device.snmpContact*. The *evt.mytimeSpan* line demonstrates using simple arithmetic on two event attributes. The two lines before the end demonstrate using a Python **method** to get values; for example *device.getSnmpStatusString()* (note the *()* at the end – this is the clue that it is a method rather than an attribute).

7.1 Using zendmd to run Python commands

So – how does one work out what attributes and methods are available? The Zenoss Administration Guide documents the **Event Database Dictionary** in Appendix D3. A **dictionary** in Python is a built-in object type for **mappings;** in other words, it is a way of holding a collection of <key> , <value> pairs. The way you access data typically, is to specify the *key* for which you require the current *value – evt.evid ,* for example, asks for the value associated with the *evid* key for the *evt* event object. Appendix D3 provides **some** of the dictionary keys available for the event object – but not all of them!

Similarly, Appendix D2 of the Zenoss Administration Guide documents attributes and methods available in TALES expressions but this information is incomplete and rather misleading (in that it says those items with parentheses after them are methods – but none of the items actually show parentheses, even though some **are** methods!)

Fortunately, Zenoss provides a Python command line interface, **zendmd**, where code for transforms can be tested out and the attributes and methods available can be explored. You should run zendmd as the zenoss user. This section is not supposed to be a Python tutorial, so some useful zendmd commands are provided in Appendix A of this paper. That said, here are a couple of tricks with zendmd.

7.1.1 Referencing an existing Zenoss event for use in zendmd

If you want to explore the attributes and methods available for an event or the device that generated the event, using zendmd, you need a way to reference an event. When executing a transform, these objects are made available to you automatically as the *evt* variable and the *device* variable – but in a zendmd test environment you need to supply these. The following figure shows a way to access an existing event in either the status or history tables of the events database. You need to supply the *evid* value by finding an appropriate event in the Event Console, bringing up the detailed data, and cutting and pasting the *evid* value into the statement in zendmd.

```
Connection to zenoss closed.
jane@bino:~> ssh zenoss
Password:
Last login: Tue Jan 13 12:53:06 2009 from bino.skills-1st.co.uk
Have a lot of fun...
jane@zenoss:~> cd /usr/local/zenoss/
jane@zenoss:/usr/local/zenoss> ./zenconsole
Welcome to Zenoss console.
bash-3.2$ su
Password:
zenoss:/usr/local/zenoss # su - zenoss
zenoss@zenoss:~>
zenoss@zenoss:~>
zenoss@zenoss:~>
zenoss@zenoss:~> zendmd
Welcome to zenoss dmd command shell!
use zhelp() to list commands
>>> evt=dmd.ZenEventManager.getEventDetailFromStatusOrHistory("0a00008337acdd92fff2ce4")
>>> print evt
<Products.ZenEvents.EventDetail.EventDetail object at 0x8c3aa8c>
>>> print evt.summary
This is a device context event
>>> print evt.device
server.class.example.org
>>>
```

Figure 22: Using zendmd to set the evt variable to an existing Zenoss event

7.1.2 Using zendmd to understand event attributes

A Zenoss event is a Python object – it is a **dictionary** data type – a data structure of <key> , <value> pairs. To see what keys (attributes) are available, use the method shown in the following figure:

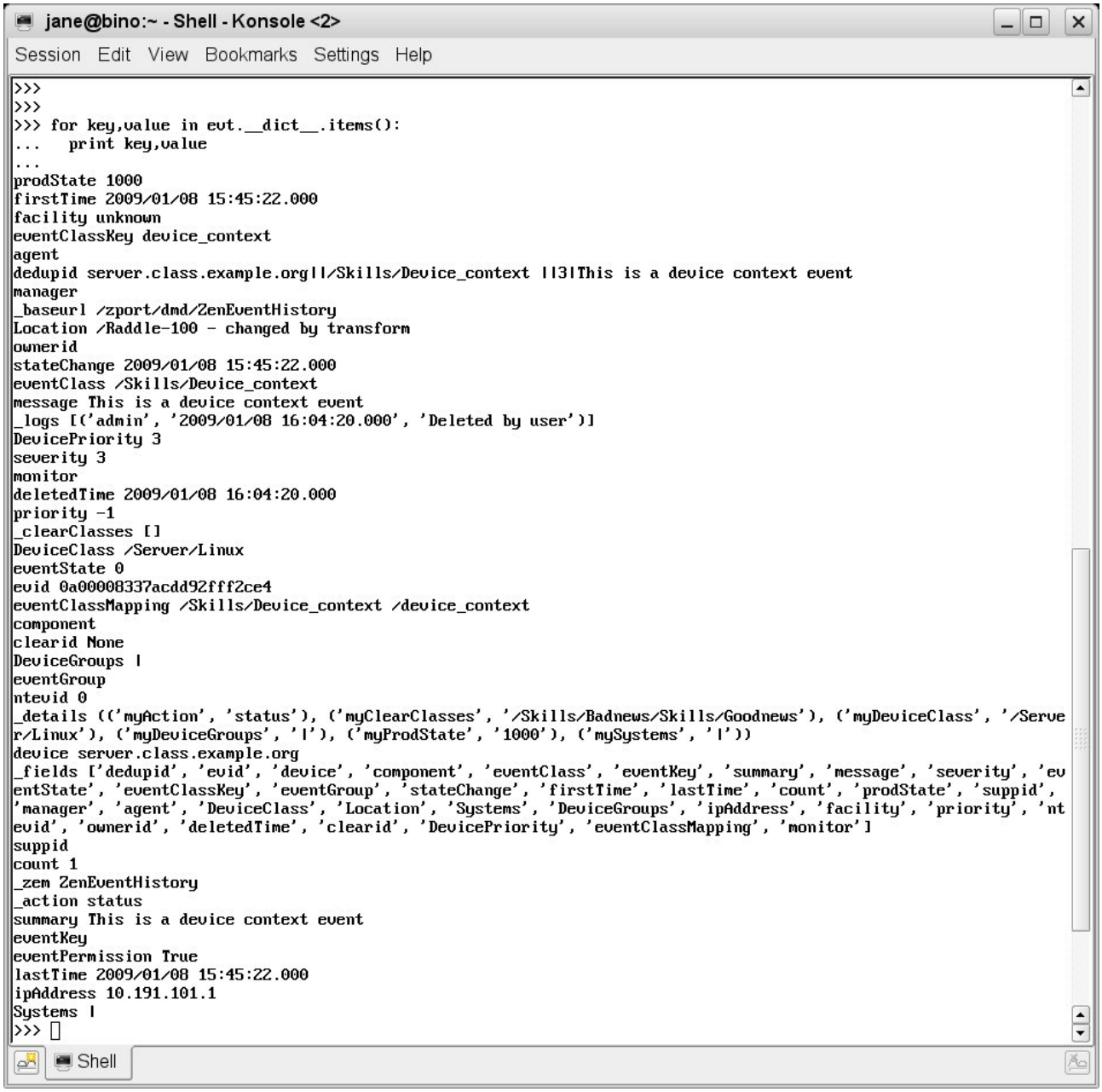

Figure 23: Using zendmd to print event attribute <key> <value> pairs

There are more attributes here than those documented in the Appendices of the Zenoss Administration Guide but they all seem to be available to event transforms. In particular, the event zProperties are available as *_action* and *_clearClasses*. You can also see the user-defined fields in the *_details* value.

Note carefully the indentation of the second zendmd statement. Python is very particular about indentation to interpret structure such as for loops. It doesn't matter how many spaces you indent the body of the for loop but it **must** be indented from the *for* line and each line in the main body of that for loop must have the same indentation. The body of a for loop, inside a for loop, would indent further – and so on.

7.1.3 Using zendmd to understand event methods

In the previous section, *evt.__dict__.items()* was used to understand the simple attributes available for the event *evt*. If you also want to understand the **methods** that are available, the *dir* function is useful:

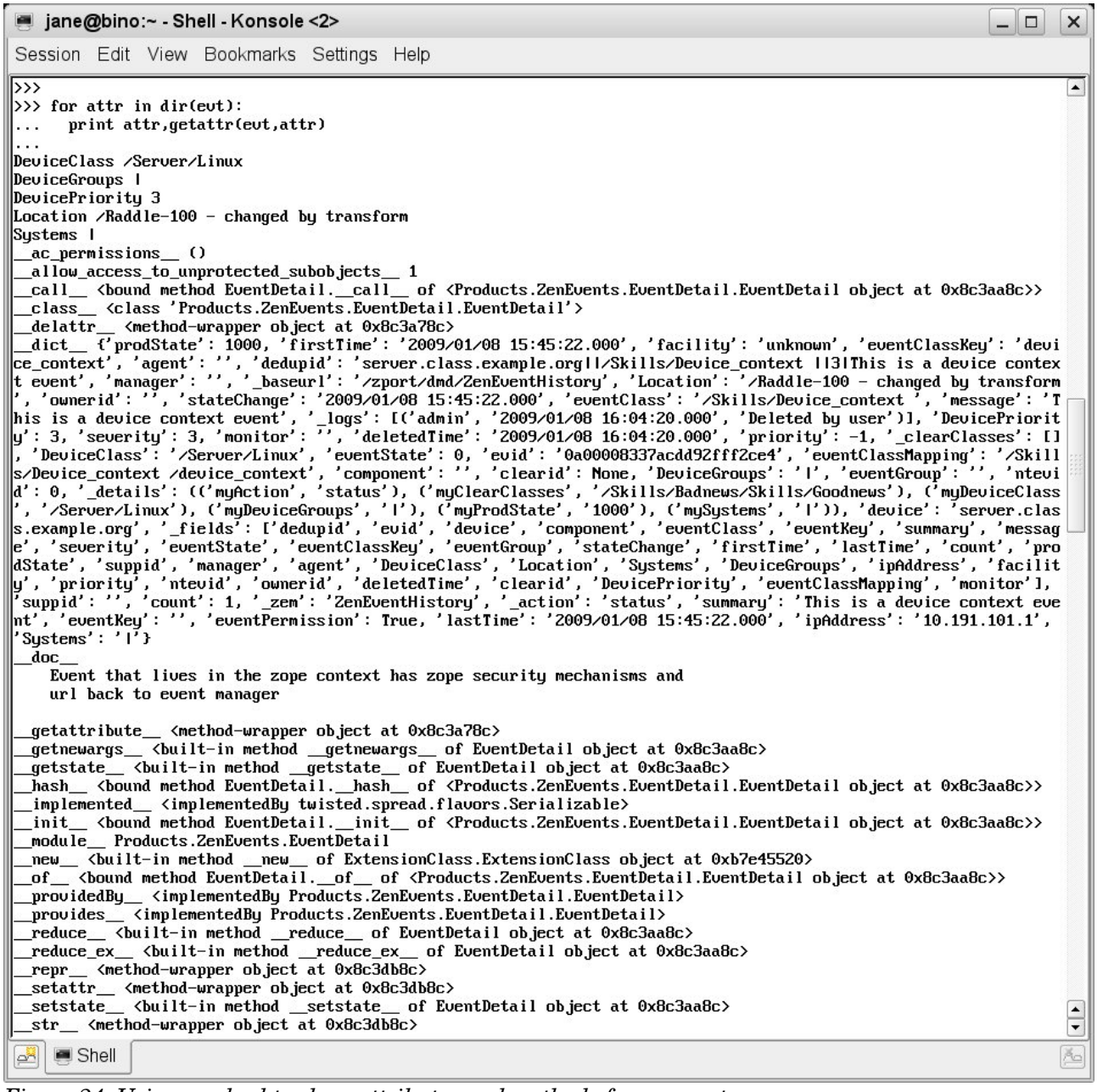

Figure 24: Using zendmd to show attributes and methods for an event

Note that this is only a partial listing!

7.2 Transform examples

7.2.1 Combining user defined fields from Regex with transform

In this example, we will return to the */Security/Su* subclass of events and combine regular expressions and transforms. The objective is, for "important devices", to escalate the event severity if a user tries to su to root but to decrease the severity if the su event comes **either** from an "unimportant" device or if the su is to a particular userid (mysql in this case). "Important" devices are determined by the event field *DevicePriority* (note two capital letters in this field name). The device priority for a device can be changed from the *Edit* tab of a device's details page.

This example is the same as shown in Figure 20 but here we focus on the transform rather than the Regex.

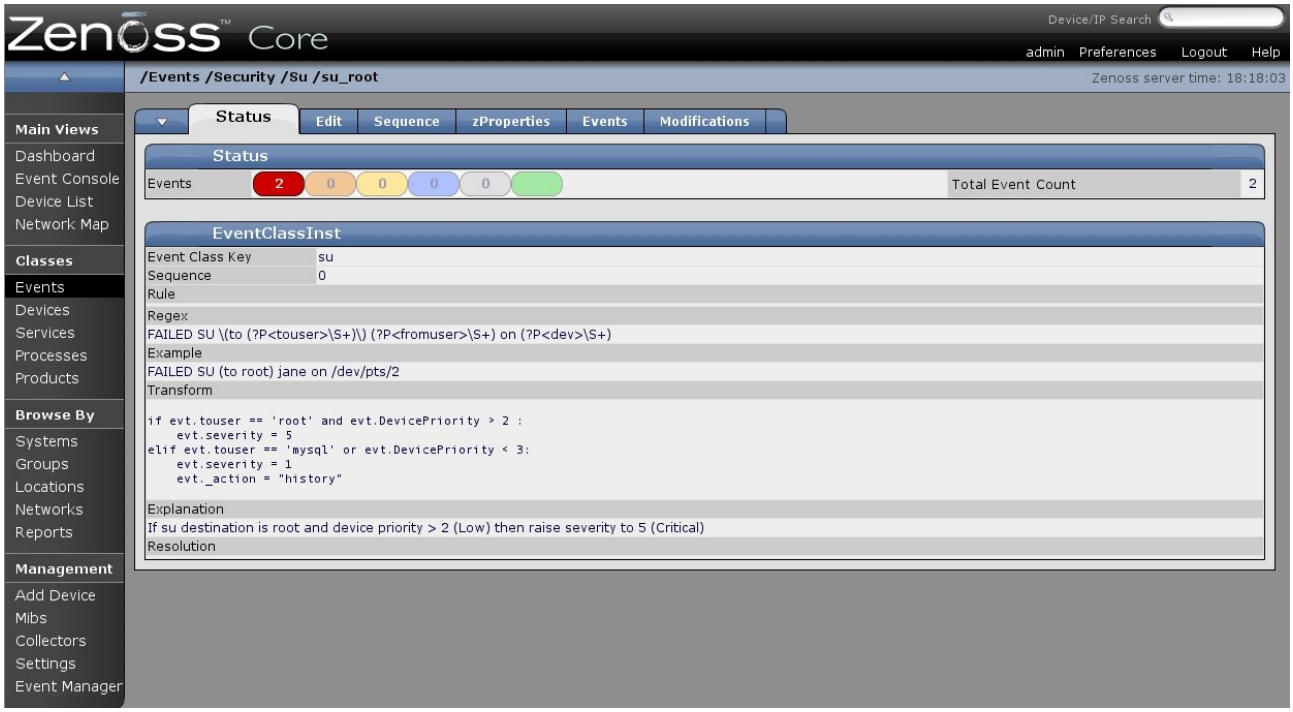

Figure 25: su_root event mapping with transform

The user-defined field *to_user*, created by the Regex, is tested against the literal string *'root'*. The result is logically ANDed with a test of the standard event field *DevicePriority* for *> 2* . If the result is True then the standard event field *severity* is set to *5* (Critical).

Again note the mandatory Python indentation for the clause following the *if* statement.

The second test in the *elif* statement is a similar test but the body of the *elif* demonstrates the ability to override the event zProperty *zEventAction* by assigning a value to *evt._action*. In this case, rather than the event being inserted into the status table of the events database, it will go directly to the history table.

7.2.2 Applying event and device context in relation to transforms

Event context is applied through the *zProperties* tab of an event class or event class mapping. **Device context** comprises the event fields **prodState, Location, DeviceClass, DeviceGroups** and **Systems**. Values for these fields are looked-up in the Zenoss database for the device that generated the event. This event mapping example demonstrates the order in which device context, event context and the mapping transform are applied.

Using the *zProperties* tab, set the *zEventSeverity* event context value to *Error* (4), zE*ventAction* to *history* and *zEventClearClasses* to */Skills*.

Test the mapping with a test event (using the *Add Event* dropdown table menu) ensuring that the severity is *Critical* (5) and the summary field is *This is a device context event* (in order to satisfy the Regex shown). The test event set the device field to *server.class.example.org* which is included in the Location called */Raddle-100* . The eventClassKey should be set to *device_context* and the eventClass should be blank.

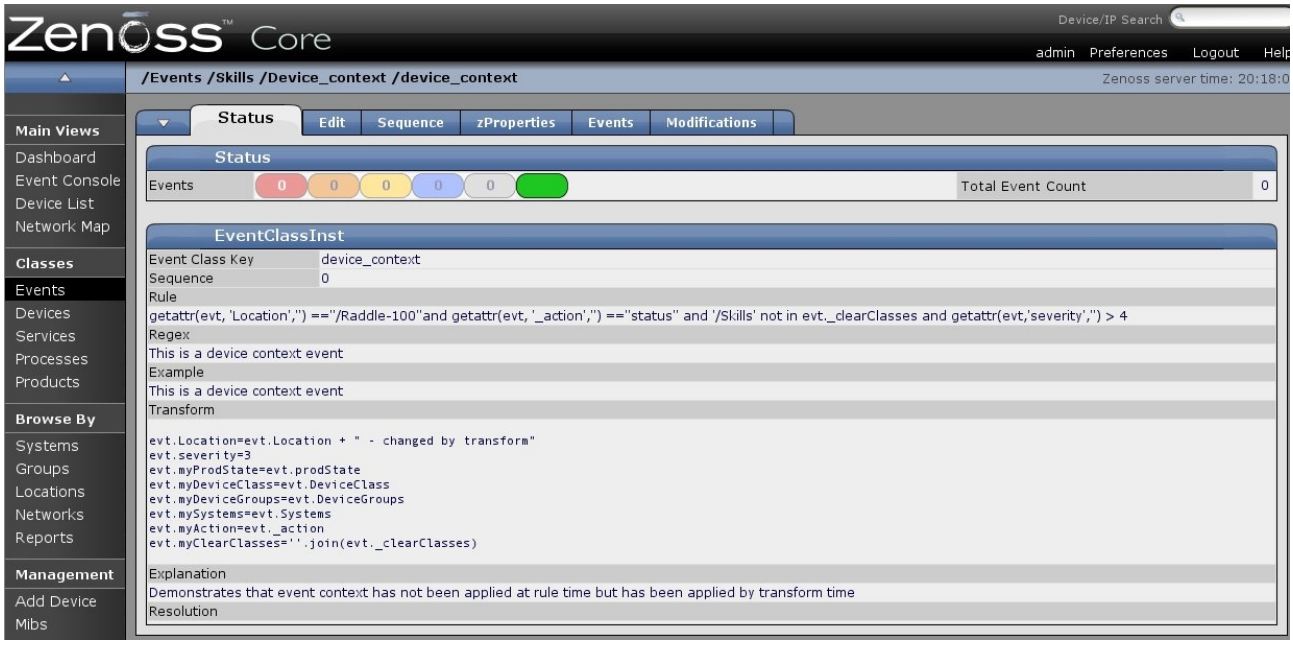

Figure 26: Combining a Rule, context and a transform for the device_context event mapping

The Rule demonstrates the Python *getattr* function to test:

- The *evt.Location* field set by device context, which should evaluate TRUE at Rule time ie. device context **has** been applied
- The *evt._action* field that is set by event context to *history*. The test shown above actually evaluates TRUE showing that event context has **not** been applied at Rule time.

- Similarly, the *evt._clearClasses* field test evaluates TRUE showing that event context has **not** been applied. The Python syntax for checking *evt._clearClasses* is a little different as this attribute is defined as a Python **list** rather than a string.
- The *evt.security* starts at 5 in the generated event and event context sets it to 4. This test evaluates TRUE confirming that event context has **not** been applied.
- **Note** that the syntax for the last field of the getattr is 2 single quotes

In summary, the Rule and Regex should evaluate successfully the transform will be applied.

The transform demonstrates:

- Changing a standard event field, *evt.Location*, to concatenate the original field value with literal text, using the "+" operator.
- Changing the evt.severity field again – it would have been modified from the original value (5) down to (4) when the event context was applied after Rule and Regex processing. The transform changes it to 3.
- Several user-defined variables are created. The evt.myClearClasses line demonstrates that all user-defined fields appear to be of type string but *evt._clearClasses* is defined as a Python list (check back with the zendmd output shown for the event directory in Figure 23. _clearClasses is followed by square brackets [ ] - this denotes a list. Strictly, referring to the same figure, the _details field is of Python type **Tuple** - which is an immutable sequence rather like a string. The round brackets ( ) denote the tuple). The bottom line is that you cannot assign evt.myClearClasses to something of type list unless you use the **join** function to stick together the list elements back into a string type.
- The user-defined fields demonstrate that both device context and event context have been applied by transform time

8 Zenoss and SNMP

8.1 SNMP introduction

The Simple Network Management Protocol (SNMP) defines Management Information Base (MIB) variables that can be polled to provide performance and configuration information. The SNMP standard also provides for agents to send “events” to a manager. Version 1 of SNMP defines these as TRAPs; version 2 of the standard calls them NOTIFICATIONs (Zenoss supports both). Both MIB variables and TRAPs / NOTIFICATIONs use Object Identifiers (OIDs) to denote different variables and events.

SNMP TRAPs are distinguished by their Enterprise Object Id (OID), the generic TRAP number and the specific TRAP number.

Natively, OIDs are defined as strings of dotted decimals that represent a path through a tree-based hierarchy, where the root of the tree is 1 and represents the iso organisation; it has a sub-branch, 3, which represents organisations (org); it has a sub-branch, 6, which represents the US Department of Defense (dod); it has a sub-branch, 1, which represents internet, and so on. Thus, all OIDs start with 1.3.6.1 .

There is a standard, **MIB-2**, which defines a number of variables that every SNMP-capable device must support; these are largely simple, network-related variables, such as interfaceInOctets. In addition to MIB-2, there are a large number of standardised MIBs defined in Request For Comment (RFC) documents; an example would be RFC 1493 defining the bridge MIB. The third category of MIBs are known as **Enterprise Specific**, which are specific to a particular vendor's particular agent – for example , the Cisco Firewall MIB. Enterprise specific MIBs often include definitions of Enterprise Specific TRAPs , in addition to MIB variables.

MIB source files translate dotted-decimal OIDs into more meaningful text. MIB files are available for many standards (like the HOST-RESOURCES MIB) and, typically, any supplier who generates their own enterprise specific MIB variables and TRAPs, should make available a source MIB file to aid this translation.

SNMP agents typically come as part of the base Operating System (Windows, Unix, Linux, Cisco IOS); however they may not be activated automatically and will require some configuration. Some agents support little more than MIB-2; others support a wide range of standard MIBs and enterprise specific MIBs.

The SNMP communication protocol varies depending on the version of SNMP. Version 1 uses a **community** name string as an authentication mechanism between SNMP manager and agent. Managers must be configured with the correct community names to use for an agent; SNMP agents must be configured for which manager(s) are allowed access to them, and which SNMP manager(s) to send TRAPs to.

In addition to requesting MIB-2 variables, Zenoss will try to access the standard Host Resources MIB to get process information for server machines. It will also attempt to access the Windows Informant MIB for all Windows server systems, in order to get CPU and file system information. The Informant MIB is a free extension subagent and MIB available from Informant at http://www.wtcs.org/informant/index.htm . Note that the base Windows SNMP agent should be installed and configured before installing the Informant extension.

Once SNMP agents are configured with community name and TRAP destination, a simple way to test them is simply to recycle the SNMP agent (indeed they will need recycling after any configuration changes). On a Windows system, use the Services utility to stop and start SNMP; on a Linux system, */etc/init.d/snmpd restart* will usually suffice. In either case you should either see a **cold start** TRAP (generic TRAP 0) or a **warm start** TRAP (generic TRAP 1) in the Zenoss Event Console. The *Details* tab should show the community name from the TRAP packet.

Another good way of generating TRAPs is to force an authentication TRAP (generic TRAP 4). An easy way to do this is to use the *snmpwalk* command with a bad community name. If the community is *public*, for a host system called *zenoss*, try:

```
snmpwalk -v 1 -c public zenoss system        test with good community
snmpwalk -v 1 -c fred zenoss system          to generate several TRAP 4's
```

8.2 Zenoss SNMP architecture

8.2.1 The zentrap daemon

zentrap is the Zenoss daemon that processes incoming SNMP TRAPs. By default, zentrap will sit on the well-know SNMP TRAP port of UDP/162 – this can be reconfigured, if required. Both SNMP version 1 TRAPs and SNMP version 2 NOTIFICATIONs are supported.

zentrap processing is implemented by the Python program *$ZENHOME/Products/ZenEvents/zentrap.py*.

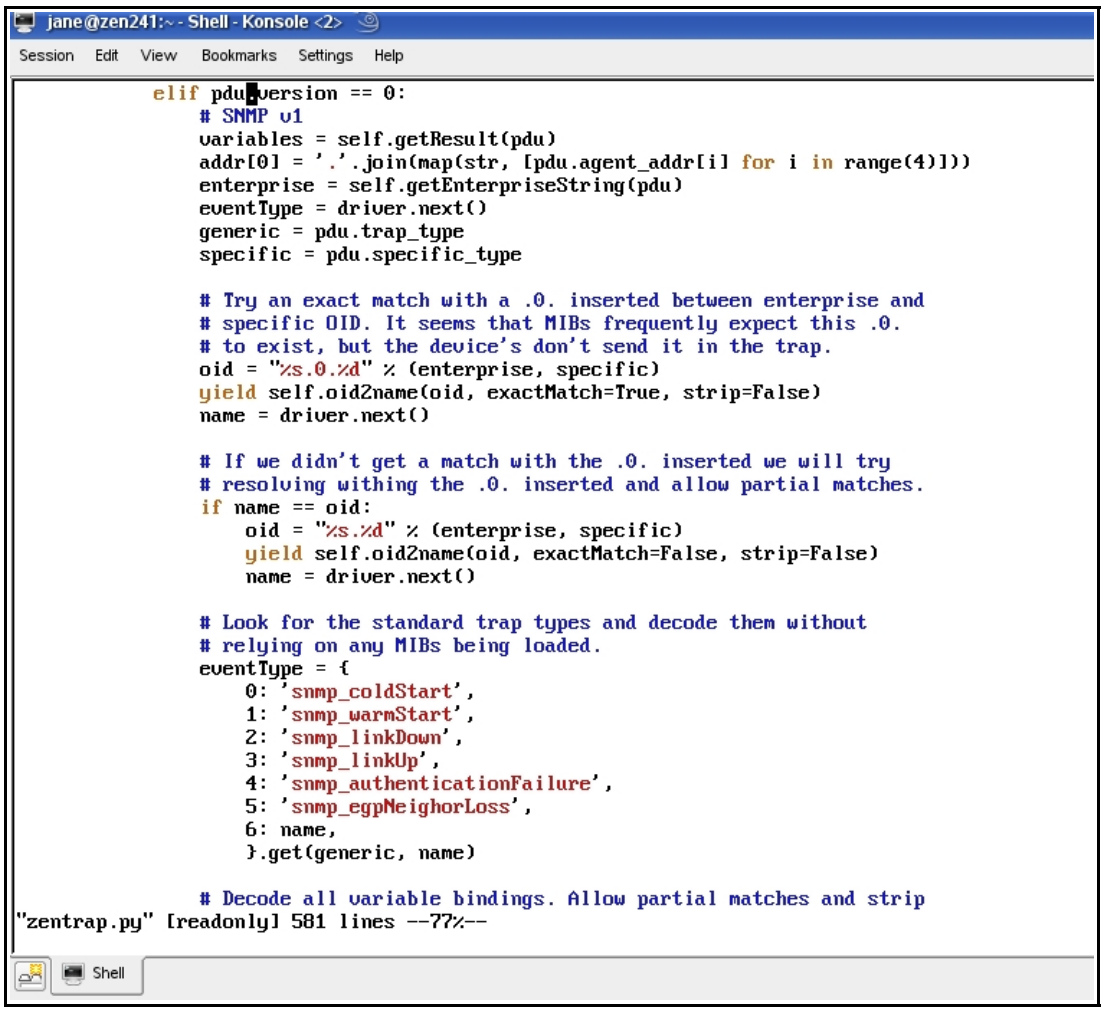

Figure 27: zentrap.py part 1 - checking for extra 0 and processing of generic TRAPs

zentrap.py parses the incoming SNMP Protocol Data Unit (PDU) to retrieve the enterprise OID, the generic TRAP number and the specific TRAP number.

The algorithm for interpreting incoming TRAP Enterprise fields has changed several times between Zenoss 2.2.x and 2.4.x because some agents have an extra 0 defined in their MIB which they do not send on an actual TRAP (see the comments in the code in Figure 27). In Zenoss 2.4.1 zentrap.py, the algorithm first tries to find a MIB in the ZODB database that corresponds with the incoming TRAP, **with** the extra 0; if this fails, then a match is searched for **without** the extra 0.

The generic TRAPs (0 through 5) are translated to strings such as *snmp_coldStart.* using the *eventType* dictionary. For specific TRAPs (generic TRAP 6), *eventType* delivers the concatenation of the enterprise OID and the specific TRAP number; for example, 1.3.6.1.4.1.123 is the enterprise, the specific trap number is 1234, so *eventType* delivers 1.3.6.1.4.1.123.1234. Any variables of the TRAP (varbinds) are also parsed out into OID / value pairs.

The *oid2name* function looks up in the ZODB database to see if translations are available for the enterprise OID, the specific TRAP number and the varbind identifiers, to translate from dotted-decimal notation to textual strings.

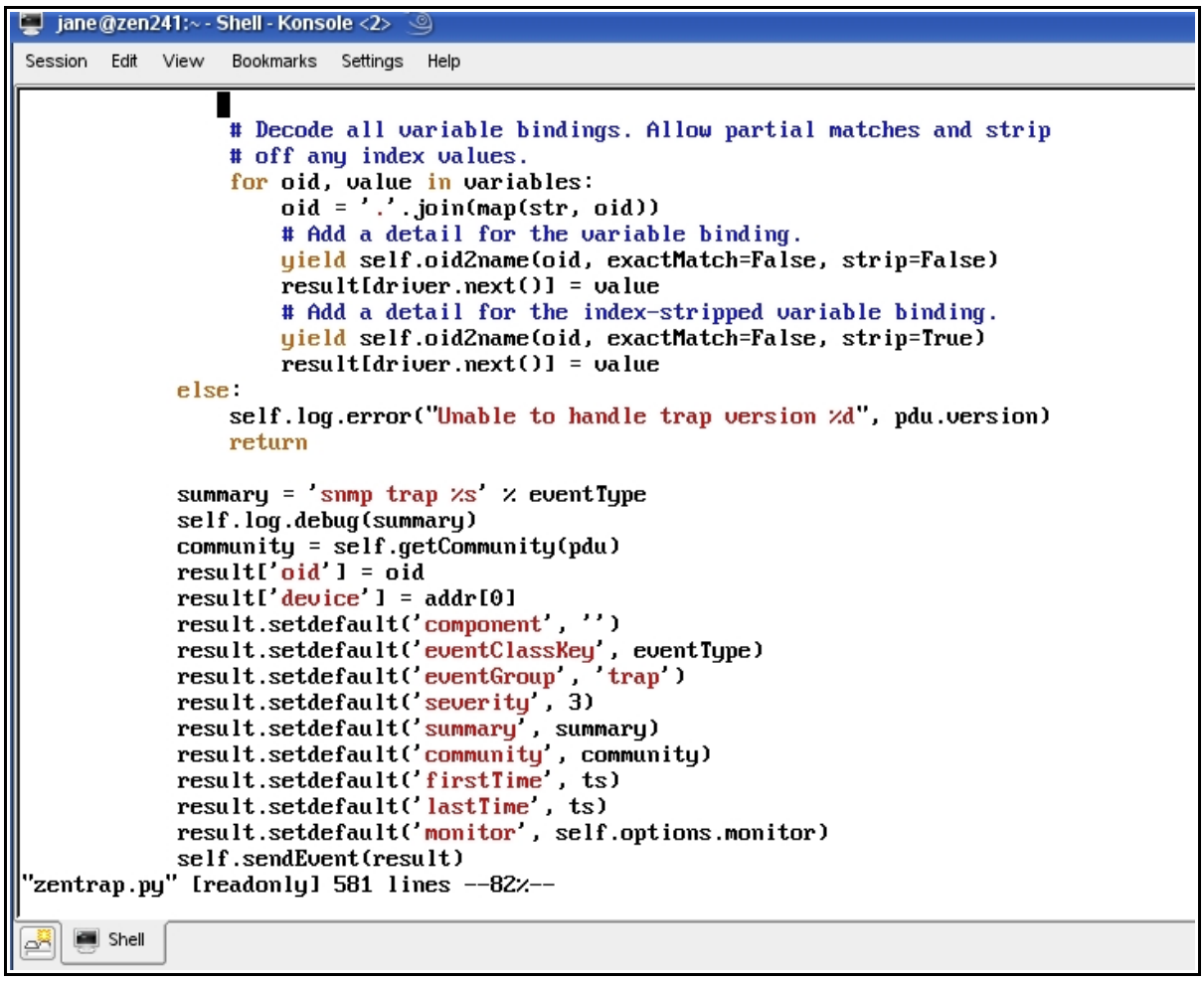

Figure 28: zentrap.py part 2 - event field settings

The following event fields are then set:

- summary *snmp trap* followed by *eventType*
- device set to the device name
- component left blank
- eventClassKey set to *eventType*
- eventGroup *trap*
- severity 3
- firstTime set to timestamp
- lastTime set to timestamp
- community set to community name string (this is a user-defined field)

8.3 Interpreting MIBs

To help decode SNMP TRAP enterprise OIDs from dotted decimal (such as . *1.3.6.1.4.1.8072.4.0.2*) into slightly more meaningful text (like *nsNotifyShutdown)* the **zenmib** command can be used to import both standard MIB source files (such as SNMPv2-SMI which defines standard OIDs) and vendor-specific MIBs. The base directory for MIBs in later versions of Zenoss is *$ZENHOME/share/mibs.*

Since many MIBs reference definitions in other MIB files via the IMPORT section at the top of the MIB file, the zenmib command needs to know where to look for prerequisite / corequisite MIBs. The environment variable *SMIPATH* should be defined to include all the standard subdirectories under *$ZENHOME/share/mibs*.

The zenmib command without parameters will try to import all MIB files that are in *$ZENHOME/share/mibs/site* . A specific MIB file can be provided as a parameter; the command should either be run from the *$ZENHOME/share/mibs* directory (in which case a full pathname is not required and the file is expected to be in that directory) or a fully qualified pathname can be specified.

8.3.1 zenmib example

To help understand the zenmib command, here is a worked example. It uses the agent for net-snmp which is the agent typically shipped with a Linux system. The enterprise OID for net-snmp is .1.3.6.1.4.1.8072.

1. Recycle a net-snmp agent with */etc/init.d/snmpd restart* . In addition to the generic cold start TRAP, you should also see TRAP .1.3.6.1.4.1.8072.4.2 . This comes from the net-snmp enterprise (.1.3.6.1.4.1.8072).
2. The actual TRAP is defined in the file *NET-SNMP-AGENT-MIB.txt* which should be shipped as part of the Operating System net-snmp package. Typically this MIB file can be found under */usr/share/snmp/mibs* . Find and examine *NET-SNMP-AGENT-MIB.txt*. Strictly, the MIB file is defining SNMP V2 NOTIFICATIONs , rather than SNMP V1 TRAPs – search in the file for the string *NOTIFI* to find the relevant lines. Also note the *IMPORTS* section at the top of the MIB file, especially the import from NET-SNMP-MIB. This indicates that NET-SNMP-AGENT-MIB is dependent on also loading NET-SNMP-MIB.

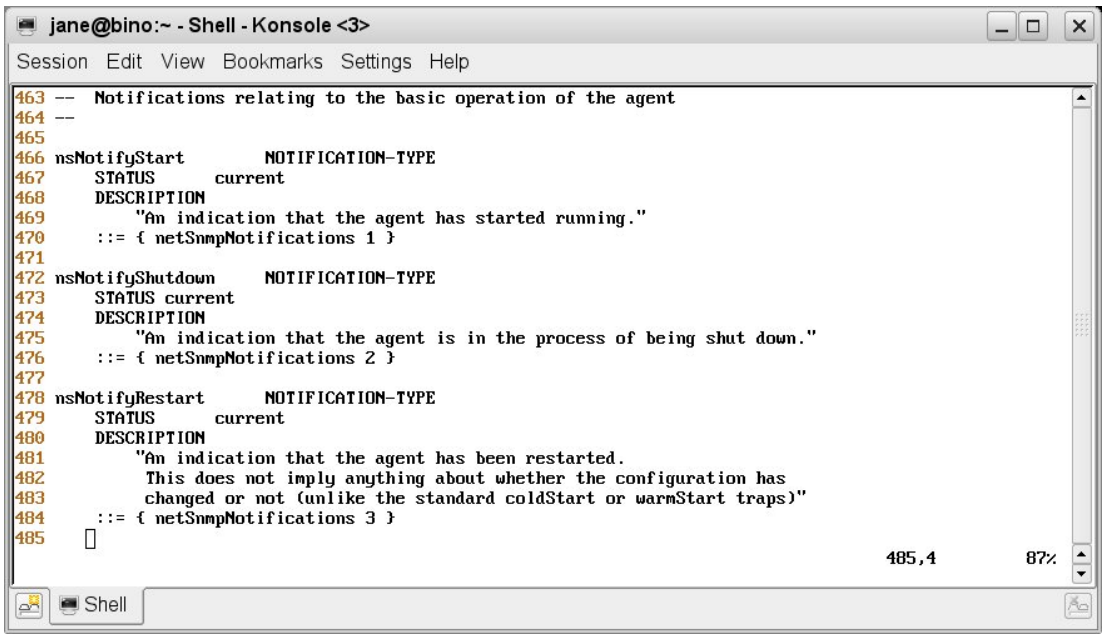

Figure 30: MIB file for NET-SNMP-AGENT-MIB showing notifications

3. Inspect the NET-SNMP-MIB.txt file and search for the string *Notifications*. You should see that the netSnmpNotificationPrefix is defined as branch 4 beneath netSnmp and that netSnmpNotifications is branch 0 under netSnmpNotificationPrefix .

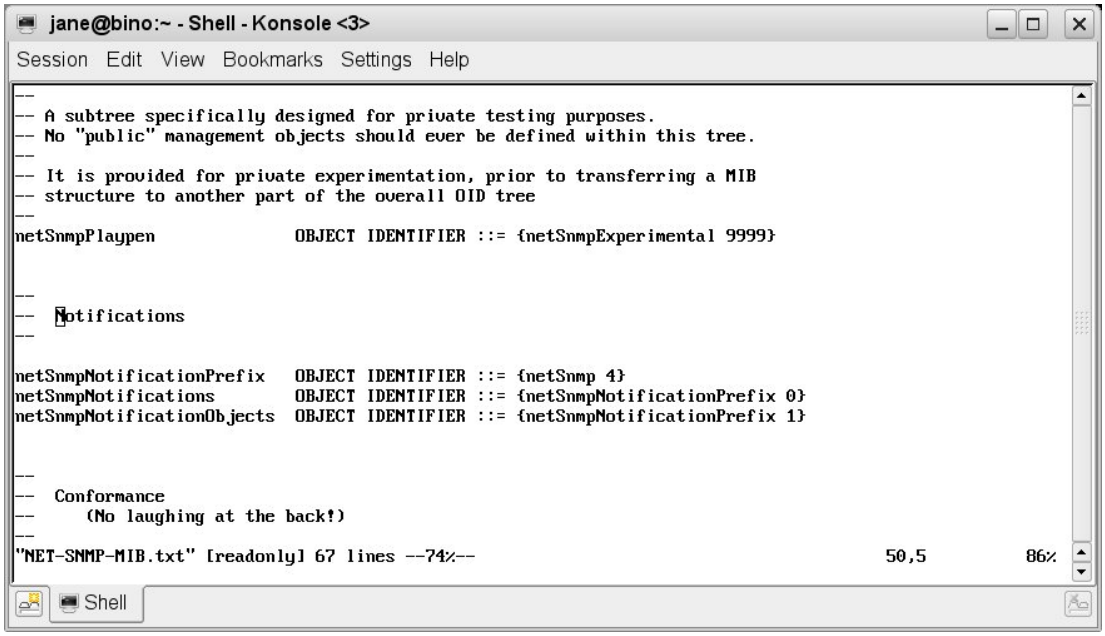

Figure 31: MIB file for NET-SNMP-MIB showing OIDs for notification hierarchy

4. At the top of the file you should find the lines that define the enterprise OID for netSnmp .

```
NET-SNMP-MIB DEFINITIONS ::= BEGIN

--
-- Top-level infrastructure of the Net-SNMP project enterprise MIB tree
--

IMPORTS
    MODULE-IDENTITY, enterprises FROM SNMPv2-SMI;

netSnmp MODULE-IDENTITY
    LAST-UPDATED "200201300000Z"
    ORGANIZATION "www.net-snmp.org"
    CONTACT-INFO
         "postal:   Wes Hardaker
                    P.O. Box 382
                    Davis CA  95617

          email:    net-snmp-coders@lists.sourceforge.net"
    DESCRIPTION
        "Top-level infrastructure of the Net-SNMP project enterprise MIB tree"
    REVISION     "200201300000Z"
    DESCRIPTION
        "First draft"
    ::= { enterprises 8072}
```

Figure 32: MIB file for NET-SNMP-MIB showing OID for netSnmp

5. Between them, these files give us (almost) the OID for the unknown TRAP we received - 1.3.6.1.4.1.8072.4.0.2 .
    - 1.3.6.1.4.1 is the standard iso.org.dod.internet.private.enterprises OID which is defined in the IMPORT from SNMPv2-SMI
    - netSnmp is {enterprises 8072 }
    - netSnmpNotificationPrefix is branch 4 under netSnmp
    - netSnmpNotifications is branch 0 under netSnmpNotificationPrefix
    - nsNotifyShutdown is NOTIFICATION 2 under netSnmpNotifications
6. Note that some SNMP agents (including the net-snmp agent) are known to omit the 0 from the TRAP that they actually generate, which is why the *oid* field in the Details tab of the event does not quite match the OID specified in the MIB file.
7. To import MIBs into Zenoss, the MIB source file needs copying to *$ZENHOME/share/mibs/site*. Copy *NET-SNMP-AGENT-MIB.txt* to this directory. At this point do **not** copy *NET-SNMP-MIB.txt*; we will demonstrate the error message when corequisite MIBs are not available.
8. To import into Zenoss use:

    ```
    zenmib run -v10
    ```

9. You should see that the *NET-SNMP-AGENT-MIB.txt* file is imported but there there should be an *INFO* tagged message saying the NET-SNMP-MIB could not be found.

10. From the Zenoss GUI, use the left-hand menu to view *Mibs*. You will probably find that the NET-SNMP-AGENT-MIB is not listed; alternatively, it may be there but if you click on it, it shows no OID Mappings and no TRAPs.
11. Copy *NET-SNMP-MIB.txt* to *$ZENHOME/share/mibs/site* and rerun the *zenmib* command. Return to the Zenoss GUI and refresh the *Mibs* menu. Clicking on the NET-SNMP-AGENT-MIB should now display a long list of OID Mappings and three TRAPs, including *nsNotifyShutdown*.
12. Restart the snmp agent on the Zenoss system with */etc/init.d/snmpd restart*. You **should** see an event in the Event Console that now contains *snmp trap nsNotifyShutdown* in the summary field, rather than *snmp trap 1.3.6.1.4.1.8072.4.2* . If this does **not** work, you may need to recycle the zentrap daemon. You can do this with the GUI from the *Settings* menu and choose the *Daemons* tab; or, as the *zenoss* user from a command line, use *zentrap restart*.
13. Zenoss has implemented a number of changes in the way MIBs are interpreted between versions 2.2.x and 2.4.x. Remember from Figure 31 that netSnmpNotifications is branch 0 under netSnmpNotificationPrefix; however, some agents omit this 0 when they actually generate TRAPs. Zenoss 2.4 now has processing in *$ZENHOME/Products/ZenEvent/zentrap.py* to try and interpret actual TRAPs both with and without the extra 0. The event console showed an event with OID 1.3.6.1.4.1.8072.4.2 for the original event; compare the Details tab of the original event with the new one that contains *nsNotifyShutdown* in the summary field. You should find that the new event has an *oid* field of 1.3.6.1.4.1.8072.4.0.2.
14. Examine *$ZENHOME/Products/ZenEvent/zentrap.py* (around line 457 for Zenoss 2.4.1) to see the code that handles this extra 0 digit processing.

8.3.2 A few comments on importing MIBs with Zenoss

There are a few quirks to do with importing MIBs into Zenoss and the quirks have changed subtly over several versions between Zenoss 2.2.x and 2.4.x.

Note that MIBs imported into Zenoss are **only** used for interpreting SNMP V1 TRAPs and SNMP V2 NOTIFICATIONs for use in the Event subsystem. Although the OIDs are imported from MIBs, they cannot be used for MIB browsing or when working with OIDs for performance sampling, thresholding and graphing.

- **Always** ensure you do MIB work as the *zenoss* user .
- The directories containing MIBs changed in 2007. They used to be under *$ZENHOME/../common/share;* with more recent versions of Zenoss they are assumed to be under *$ZENHOME/share*. To ensure MIB importing works run the following commands to establish symbolic links:

```
cd $ZENHOME
ln -s /usr/local/zenoss/common/share
ln -s /usr/local/zenoss/common/libexec
```

- MIB importing often requires pre-requisite MIBs to also be imported. Zenoss provides many of the standard MIBs in the *ietf, iana* and *irtf* subdirectories of *$ZENHOME/share/mibs*. To automatically find these pre-requisite MIBs, an environment variable, *SMIPATH* is required which includes each of the directories under *$ZENHOME/share/mibs*. The best way to ensure this is to modify the *.bashrc* file for the zenoss user and include the variable there (all on one line):

```
export SMIPATH=$ZENHOME/share/mibs/iana:$ZENHOME/share/mibs/ietf:$ZENHOME/share/mibs/irtf:$ZENHOME/share/mibs/site:$ZENHOME/share/mibs/tubs/
```

- By default. *zenmib run -v10* will try and import everything under *$ZENHOME/share/mibs/site*. The *-v10* simply adds more verbose output. *zenmib* should check in the other directories (in $SMIPATH) for prerequisites. **Sometimes this just doesn't seem to work!**
- Whenever you have imported a MIB, check at the GUI on the *Mibs* page. You should see the name of the MIB **and** you should usually see non-zero counts under the *Nodes* and/or *Notifications* columns.

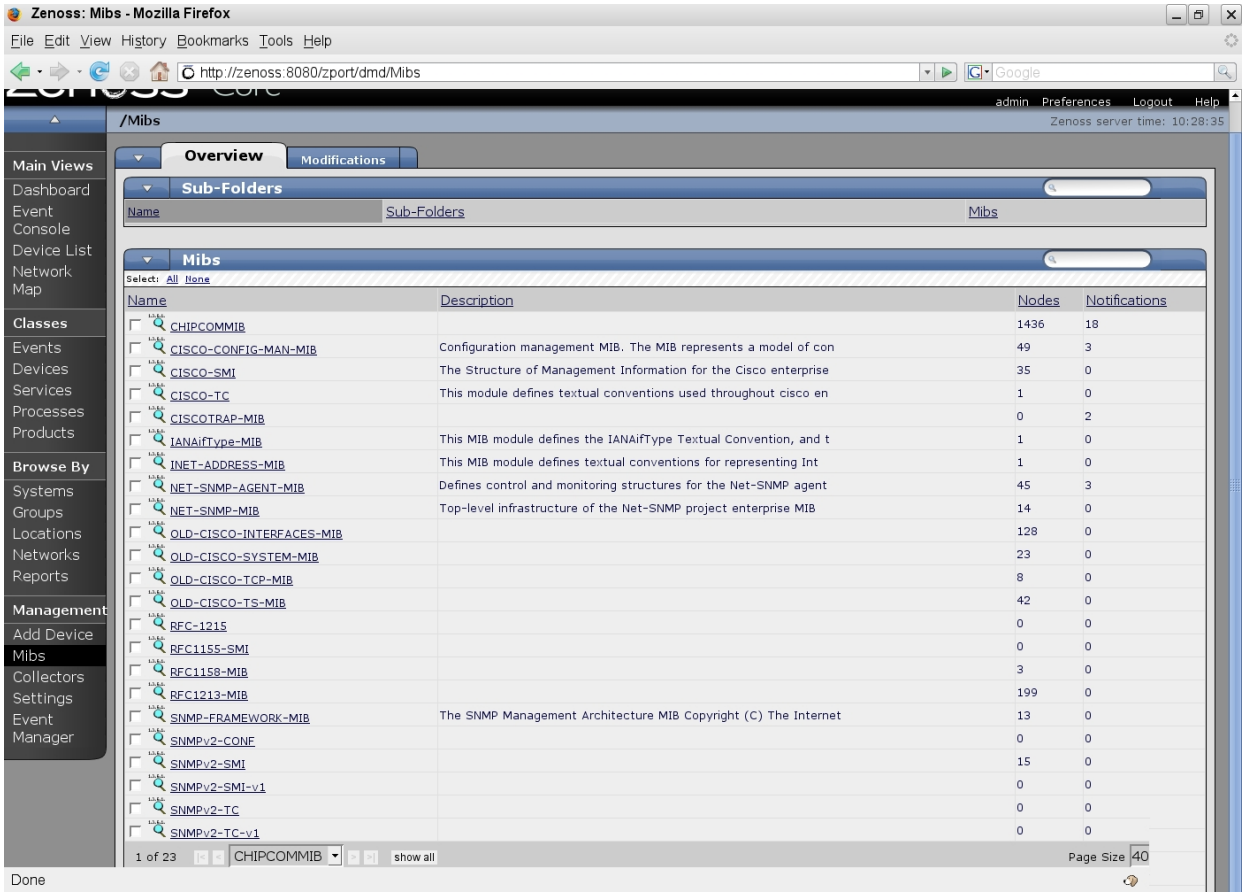

Figure 33: /Mibs page showing successfully imported MIBs

- There are some MIBs that **will** result in zero counts, for example if the MIB source file only defines SNMP structure and does not include the definition for any OIDs (that get translated under the *Nodes* column) or any TRAPs or NOTIFICATIONs (that get translated under the *Notifications* column). If you import a MIB and get zeros in both columns, check the source file of the MIB to see whether there should be entities.
- Check the output of the zenmib command carefully for error messages.
- If you get an error message or you get zero counts in the GUI, you should see in the verbose output of the *zenmib* command that the *smidump* command is being run with error output sent to */dev/null*. As a debugging tool, copy and paste the *smidump* line but omit the ending *2>/dev/null* . This should show more information on what is going wrong with the mib definitions and pre-requisite chains of IMPORTs. **Note** that the *smidump* command is only checking MIB viability; it is **not** doing anything to actually import the MIB into Zenoss. *smidump* must specify all its prerequisites in the correct order.

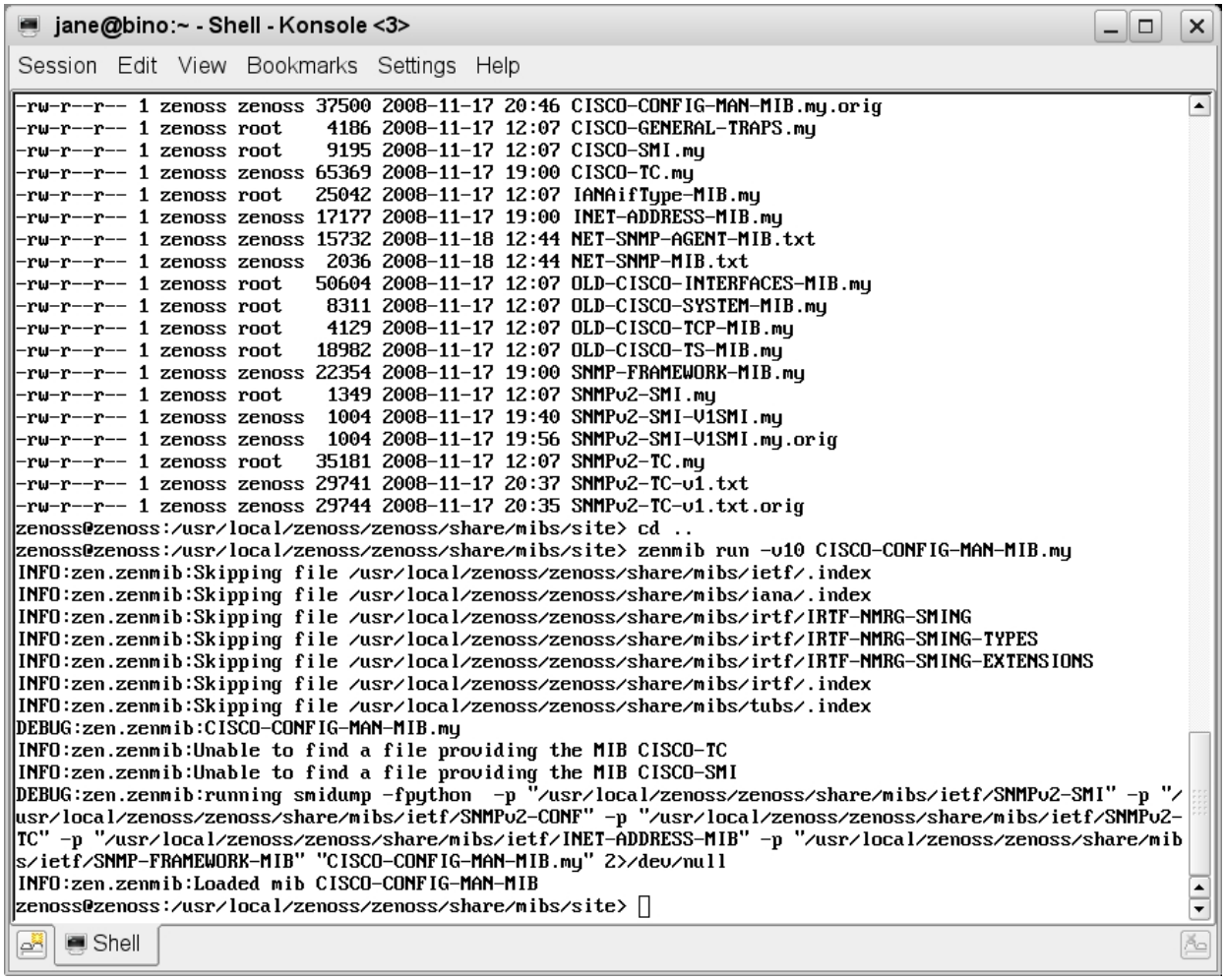

Figure 34: zenmib run command with missing IMPORT chain

- Note the messages about missing CISCO-TC and CISCO-SMI even though they have been imported previously
- Also note the *smidump* command that can be cut and paste, omitting *2>/dev/null* for extra debugging. If the *smidump* ends in listing the MIB, rather than error messages, then the problem is not really with the MIB itself or it's IMPORT chain.
- Also note the *smidump* checks the MIB to be compiled for IMPORTs and has automatically found the prerequisites (*-p* parameters).

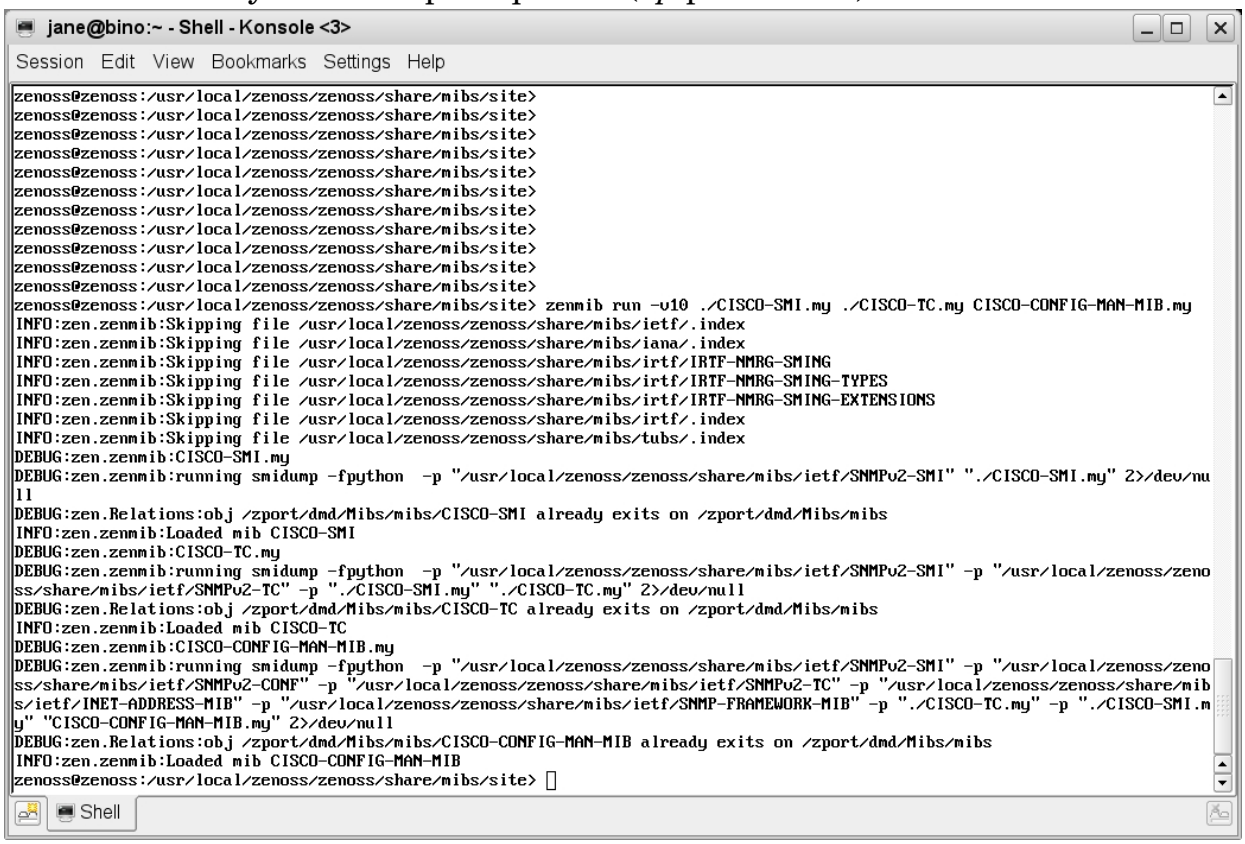

Figure 35: zenmib run command specifically including prerequisite files

- Even if you have already imported a prerequisite MIB successfully, the Zenoss database does not know about the IMPORT chains; it only has OIDs and TRAPs / NOTIFICATIONs. For this reason you will sometimes need to run the *zenmib* command with any missing prerequisites on the *zenmib* line, **ahead** of the mib that you actually want to import.
- Note that the prerequisite files that have already been loaded get a DEBUG comment that says it “already **exits**” - I suspect this should say “already **exists**”!
- It is always possible to specify a full path name to a MIB file. However, if you make an error in typing filename or path then you will get a “Failed to locate mib...” message from the *smidump* command.
- Before attempting to reimport a MIB, select the MIB and use the table menu from the *Mibs* panel of */Mibs* and select *DeleteMibs*.

- Beware that some MIBs have a rather different MIB name (on the first line of the MIB source) than the filename that contains them. This provides ample opportunity for confusion!
- Beware that some MIBs are wrong! For example, the CISCO-CONFIG-MAN MIB uses a type of Unsigned64 which is not legal in its context. You can make this MIB import by editing the source file to change both occurrences of *Unsigned64* to *Unsigned32*.
- Good sites to look for Cisco MIBs and their prerequisites are:
  - http://tools.cisco.com/Support/SNMP/
  - ftp://ftp-sj.cisco.com/pub/mibs/ including V1 versions of V2 SNMP MIBs
- Other tricks to try, found from the Zenoss fora:
  - If you cannot see imported MIBs or those MIBs can be seen, have non-zero counts but don't translate incoming events, try the following commands from within *zendmd* :

```
dmd.Mibs.reIndex()
commit()
```

  - To check from within *zendmd* what MIBs are loaded (note that the second line must be indented):

```
for mib in dmd.Mibs.mibs():
    print mib
commit()
```

  - Try recycling the *zentrap* command from the *Daemons* tab of the *Settings* menu or, as the *zenoss* user, run *zentrap restart*.

8.4 The MIB browser ZenPack

There is an excellent community ZenPack available to perform MIB Browsing. This is not directly relevant to TRAP / NOTIFICATION processing, but it is useful for investigating MIBs with a view to building SNMP performance templates. It can be downloaded from http://www.zenoss.com/community/projects/zenpacks/mib-browser .

It provides a MIB browser to explore any OID that has been loaded into Zenoss, along with a test facility to *snmpwalk* a configurable device to retrieve values for any selected part of the MIB tree.

8.5 Mapping SNMP events

Zenoss provides some event mappings for SNMP TRAPs out-of-the-box. As discussed in an earlier section, the file *$ZENHOME/Products/ZenModel/data/events.xml* configures all the standard mappings so searching this file for SNMP provides insight for default customisation.

Most SNMP TRAPs map to the Zenoss **/Unknown** event class. There are one-or-two exceptions for some generic TRAPs such as Link Up (3), Link Down (2) and the Authentication TRAP (4). Event fields that are automatically populated by the **zentrap** processing include *summary, eventClassKey* and *agent*. The *Details* tab of the event detail shows the **community** and **oid** Field / Value pairs (the oid field is a recent addition to the *Details* tab).

This means that, typically, the event only maps on the Event Class Key, which is interpreted by zentrap.py as *enterprises.<enterprise number>.<specific trap>* if the SNMPv2-SMI has been imported or *1.3.6.1.4.1.<enterprise number>.<specific trap>* otherwise. The *summary* field will be *snmp trap <enterprise OID><specific trap>* and the *agent* field will be set to *zentrap* .

TRAPs and NOTIFICATIONs may have one or more TRAP variables (varbinds). These varbinds appear in the *Details* tab of an event detail where the field name is the varbind OID and the corresponding field value is the value of that varbind. Event class mappings can be devised with various Rule, Regex and Transform elements, to parse out the intelligence from SNMP TRAPs and either create new user-defined event fields or modify existing fields (such as *evt.summary*).

Note that event mappings that parse out SNMP OIDs and varbinds must be aware of whether the relevant MIBs have been imported, or not. If a MIB is imported, OID mapping based on matching dotted-decimal notation will fail as the MIB OID translations happen **before** event mapping.

8.5.1 SNMP event mapping example

In order to interpret enterprise specific TRAPs, mappings are usually required. Often an action or modification is required, effectively based on what enterprise the TRAP came from (Cisco, net-snmp, ...), so a subclass of events are required that inherit some common characteristics but some event details vary depending on the exact enterprise specific TRAP number.

Many enterprise TRAPs also include several varbinds that need to be interpreted and processed.

In the mapping example shown here, three small scripts are used to generate TRAPs from the 1.3.6.1.4.1.123 enterprise – one for each of specific TRAPs 1234, 1235 and 1236. The first two have a single varbind whose string-type value is “Hello world 4”, where the end number is 4 or 5; the third script generates a TRAP with 2 varbinds. Note that each of the varbinds exhibit the “extra 0” behaviour, ie. the varbind field will be 1.3.6.1.4.1.123.0.1234.

1. Without any mapping, when *gen_mytrap_1234.sh* is run, it will map to the */Unknown* event class.
2. Create a new event subclass *Snmp* under the class */Skills* .

3. Map the “1234” event by selecting its tick-box and using the table menu to *Map Events to Class* . Choose */Skills/Snmp* from the dropdown selection box. Leave the rest of the Event Class Mapping parameters as defaults for now. This means that the event only maps on the eventClassKey, which translates to *<enterprise OID>.<specific trap>* . The mapping name is automatically assigned the name of the eventClassKey (*1.3.6.1.4.1.123.1234* if SNMPv2-SMI is **not** imported; *enterprises.123.1234* if it is). Refer back to the snippet of the zentrap code in Figure 33 for more information on the parsing of the TRAP into event fields. Check that your event class mapping works.

The next step is to interpret the varbind. Each of the TRAPs generated by the test scripts come from the Enterprise 1.3.6.1.4.1.123 and hence each of the varbind fields in the Details tab will start with 1.3.6.1.4.1.123. A transform will extract that part of the OID **after** 1.3.6.1.4.1.123 . It will also substitute the value of the varbind into the event summary.

1. Return to your Event Class Mapping called *1.3.6.1.4.1.123* under */Events/Skills/Snmp* .
2. Edit the *Transform* box:

```
for attr in dir(evt):
    if attr.startswith('1.3.6.1.4.1.123.'):
        evt.myRestOfOID=attr.replace('1.3.6.1.4.1.123.','')
        evt.myFieldValue=getattr(evt,attr)
        evt.summary=(evt.summary + “  “ + evt.myFieldValue)
```

3. The *dir(evt)* syntax provides a list of the methods and attributes available for the current event, *evt* .
4. The “startswith” line ensures that transforms only take place with attributes that start with 1.3.6.1.4.1.123 – ie. varbind attribute fields.
5. Note that the “replace” line is replacing the OID specified, with the null string – the syntax after the comma is single-quote single-quote . The rest of the attribute (ie. the 0.1234 bit) is kept and becomes the value of the user-field myRestOfOID .
6. The “getattr” line gets the value of the given attribute of the event.
7. Running the script to generate a “1234” TRAP should now generate an event with:
    - The event mapped to the */Skills/Snmp* class
    - The summary field should say *“snmp trap 1.3.6.1.4.1.123.1234 Hello world 4”*.
    - The *Details* tab of the detailed data should show values for *community*, *oid*, *myFieldValue* and *myRestOfOID*, in addition to the default varbind field/value pair of *1.3.6.1.4.1.123.0.1234 / Hello world 4*

8. Running the script to generate a “1235” TRAP will still generate an event with the */Unknown* class as the event class mapping is based on the eventClassKey of *1.3.6.1.4.1.123.1234* .

So far, we are only matching a single SNMP TRAP with the eventClassKey field. The objective is to map all events from the enterprise 1.3.6.1.4.1.123 . With SNMP, you often want to apply a transform to several similar events which are often distinguished by the later parts of the OID field. The test scripts all generate events whose eventClassKey start with *1.3.6.1.4.1.123.* but they differ in the last number.

A **Rule** will be used to match all appropriate events. However, a Rule is only inspected if the eventClassKey has already matched successfully and we have no control over the eventClassKey – that is set by *zentrap.py* . Thus, the **defaultmapping** concept will be used.

1. Clear all SNMP events for your Zenoss system.
2. Edit the *1.3.6.1.4.1.123.1234* mapping.
   - In the *Rule* box put *evt.eventClassKey.startswith('1.3.6.1.4.1.123.')*
   - Change the *Name* of the mapping to *1.3.6.1.4.1.123*
   - Save the mapping away
3. Run the *gen_mytrap_1234.sh* script and the *gen_mytrap_1235.sh* script.
4. Check the events in the Event Console
5. You should find that the 1234 TRAP maps successfully but the 1235 TRAP doesn't. This is because the initial test for event class mapping checks the eventClassKey – that is still set to *1.3.6.1.4.1.123.1234* so the processing never even gets as far as checking our Rule! **Note** that we have no control over how the eventClassKey field is populated by the event processing mechanism – it is parsed out for us by *zentrap.py* (see Figure 33 again).
6. This is where the “magic string” of *defaultmapping* can be used in the Event Class Key field. Set the Event Class Key to *defaultmapping* (N**ote** it **must** be all lower case). If the process of mapping an event cannot find a match for the Event Class Key then it will re-run the mapping process with an Event Class Key of *defaultmapping*.
7. Save the mapping.
8. Open the *Sequence* tab. There are several mappings that all map on an Event Class Key of *defaultmapping*. Choose a suitable sequence number for the new *defaultmapping*. Save the mapping.
9. Clear existing events. Rerun both scripts. Check that both events now map correctly.

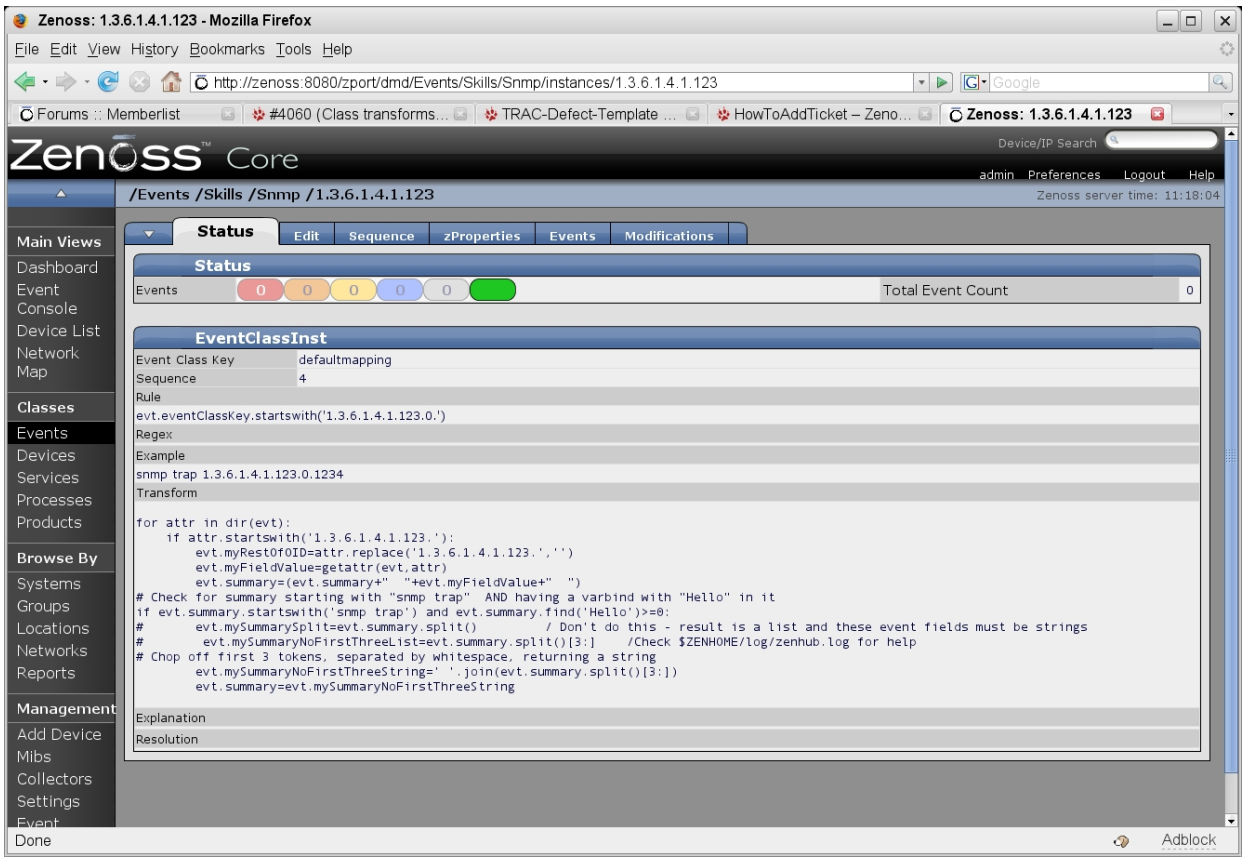

Figure 36: Mapping for SNMP TRAP with rule, transform and eventClassKey of defaultmapping

The test events used so far, only have one varbind. What if your TRAP has several varbinds and you want to use information from each of them? The script *gen_mytrap_1236.sh* generates a specific TRAP 1236, with two varbinds:

- varbind 1 1.3.6.1.4.1.123.0.12361 Hello world varbind1 61”
- varbind 2 1.3.6.1.4.1.123.0.12362 Hello world varbind1 62”

Running the script *gen_mytrap_1236.sh* should result in an event that maps to the */Skills/Snmp* class, with the *myFieldValue* and *myRestOfOID* fields matching the data in the **last** varbind and the summary also reflecting only the data in the **last** varbind. To make the mapping take account of both varbinds:

1. Edit the *1.3.6.1.4.1.123* mapping and examine the *Transform* box. Note that it starts with *for attr in dir(evt):* . This construct will loop around **all** event fields. The second line will select just the field names that start with *.1.3.6.1.4.1.123.* .
2. To construct an event summary that contains information from **each** varbind event field, change the *evt.summary* field to:

   ```
   evt.summary=(evt.summary+”  “+evt.myFieldValue+”  “)
   ```

3. Clear old SNMP events and run *gen_mytrap_1236.sh* . Your summary field should now reflect the data in all of the TRAP varbinds.

Suppose you now wanted to modify the summary field so that, if it started with the standard “snmp trap” followed by the OID, **and** the summary field now has had appended a number of varbinds, at least one of which contains the string “Hello”, then chop the *snmp trap OID* bit off the front of the summary.

Here is a transform that uses the Python *split* and *join* methods to do that.

Transform

```
for attr in dir(evt):
    if attr.startswith('1.3.6.1.4.1.123.'):
        evt.myRestOfOID=attr.replace('1.3.6.1.4.1.123.','')
        evt.myFieldValue=getattr(evt,attr)
        evt.summary=(evt.summary+"  "+evt.myFieldValue+"  ")
# Check for summary starting with "snmp trap"  AND having a varbind with "Hello" in it
if evt.summary.startswith('snmp trap') and evt.summary.find('Hello')>=0:
#       evt.mySummarySplit=evt.summary.split()         / Don't do this - result is a list and these event fields must be strings
#        evt.mySummaryNoFirstThreeList=evt.summary.split()[3:]    /Check $ZENHOME/log/zenhub.log for help
# Chop off first 3 tokens, separated by whitespace, returning a string
        evt.mySummaryNoFirstThreeString=' '.join(evt.summary.split()[3:])
        evt.summary=evt.mySummaryNoFirstThreeString
```

Figure 37: Event transform for SNMP TRAP with split and join

The second *if* clause demonstrates the Python *find* function to search for the string “Hello” (it returns the offset in the string of the substring you specified (the first character has position 0) - if the substring doesn't exist then *find* returns *-1*).

evt.summary.split()[3:] splits the evt.summary field into a **list** of strings and the *[3:]* on the end selects everything after the third element in the list. The result is that the substrings “snmp” , “trap” and the OID are chopped off the beginning of *evt.summary*. The problem is that the resulting data structure is still a **list** of strings and *evt.summary* is defined as a **string**, not a list. To rectify this, the Python *join* function is used to convert our summary back into a string.

Check the end of *$ZENHOME/log/zenhub.log* and *$ZENHOME/log/event.log* for debugging help.

9 Event Commands

When an event occurs, it is possible to run a command on the Zenoss server. The fields of the event and the attributes of the device that generated the event, are made available to use in the event command. Event commands are shellscripts (though they can call other programs such as Python scripts). It is possible to further customise such commands to build in a delay before execution and to run the script at repeated intervals.

Event Commands have access to the event fields and device attributes, using TALES expressions (**T**emplate **A**ttribute **L**anguage **E**xpression **S**yntax, from Zope) to reference fields of the event, *evt* and attributes of the device, *dev*. See Appendix D of the Zenoss Administration Guide for more details. **Note** that you **must** use TALES – the *evt.<event field>* syntax used in mapping rules and transforms does not work in event commands. TALES syntax takes the form:

```
${evt/<event field>}
${dev/<device attribute>}
```

9.1 Creating event commands

Event commands are created using the left-hand *Event Manager* menu, and then select the *Commands* tab.

Figure 38: Creating new event commands

Type the new command file name in the box at the bottom of the panel and click *Add*. Click on the new name that you have just created to define the command.

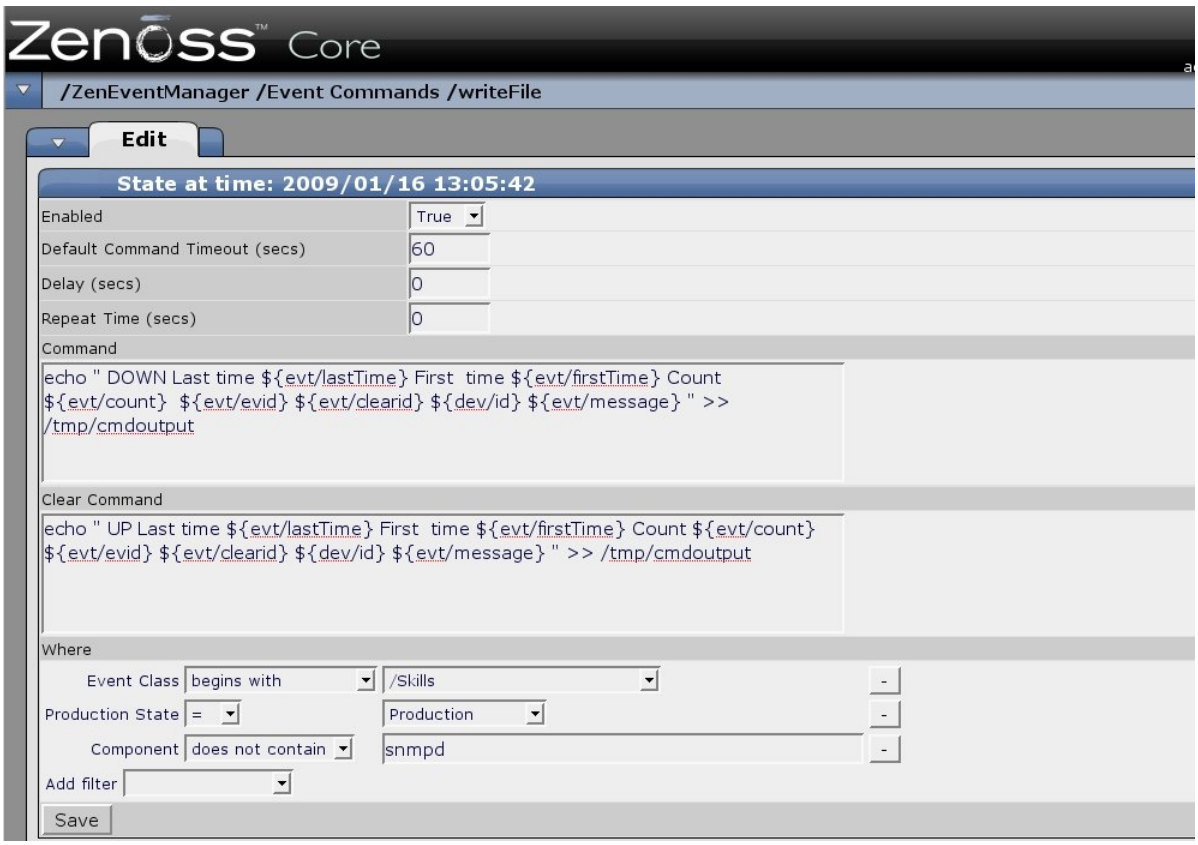

Figure 39: Defining a simple Event Command called writeFile

The dialogue to configure an event command includes:

- An enabled flag
- A timeout field for the command
- A delay field such that a delay can be built-in between the event occurring and the command running. For "glitch" scenarios, an event may be cleared rapidly so an automation script should be delayed pending that possibility.
- A repeat field that denotes how frequently to repeat the command
- Different commands can be run depending on whether the event is reporting a problem or whether the event is being cleared (moved to the history table of the events database).
- Through the use of one or more filters, it is possible to be very specific about what events should result in commands being run. Filters should be used judiciously or performance degradation is likely to take place if event commands run on most events.

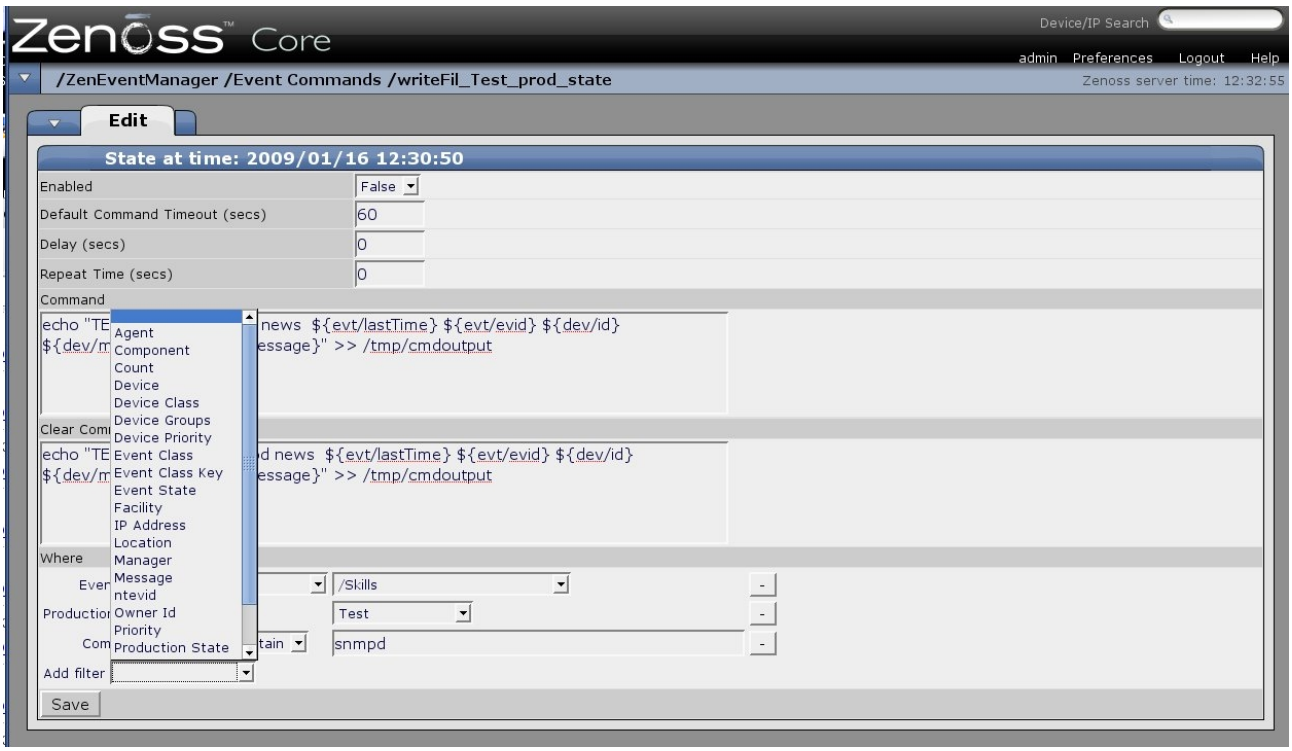

Figure 40: Event command dialogue with filter options dropdown

Filters for an Event Command are logically ANDed. To implement a logical OR of criteria, create one or more enabled event commands. Each and every command that satisfies its filtering criteria, will be executed. If multiple commands are valid, they will all be run.

Event commands are executed asynchronously by the **zenactions** daemon which, by default, runs every minute. Commands will be run **once** for **all** open events in the status table of the events database, provided the command has not yet been run. The **alert-state** table of the events database records whether a command has been run or not for a particular command, for a particular event.

zenactions also runs any alerting rules (discussed in the next section). It's logfile is *$ZENHOME/log/zenactions.log*.

9.2 Debugging event commands

You will find that duplicate events do **not** run event commands. If logging for zenactions is increased, it is possible to understand what is going on. To increase the debugging level of *zenactions* to *Debug (10)*, use the left-hand *Settings* menu and the *Daemons* tab. Refer back to Page 27 where this procedure was discussed for increasing logging for the zensyslog daemon. You will need to recycle the zenactions daemon for this to take effect.

In the example below, the event command is called *writeFile* . It logs the event fields *firstTime* , *lastTime* and *count* to an output file, for both commands and clear commands,

for both “goodnews” and “badnews” events. Initially, a “badnews” event is generated, followed by a **duplicate** “badnews” event, followed by a **different** “badnews” event.

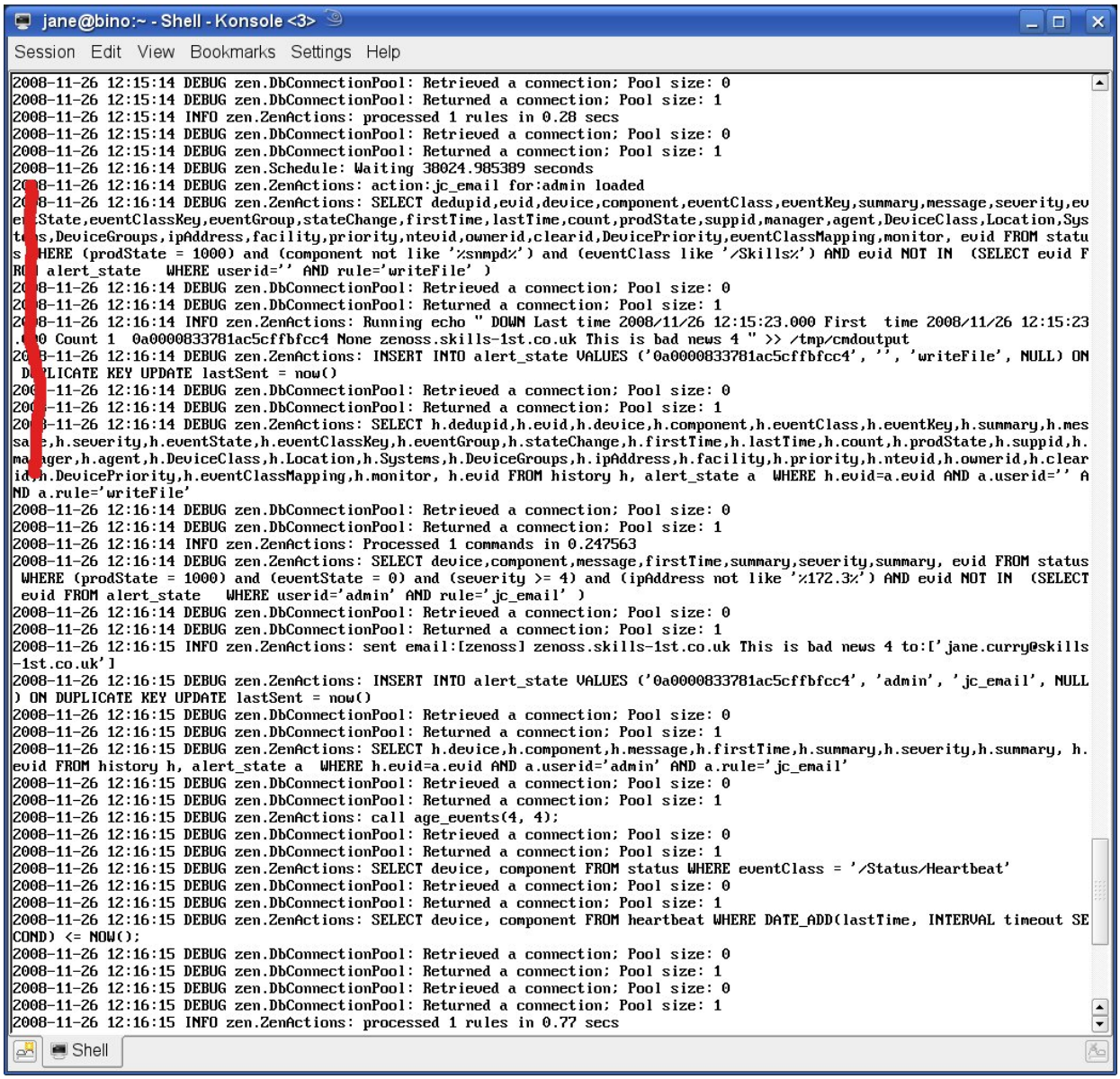

Figure 41: zenactions.log showing actions for writeFile

On examination of zenactions.log:

- You should see zenactions waking up every minute to process commands
- You should see a SELECT on the status database relevant to writeFile
- You should see the echo command being executed
- You should also see a line similar to:
  - INSERT INTO alert_state VALUES ('0a0000833781ac5cffbfcc4', '', 'writeFile', NULL) ON DUPLICATE KEY UPDATE lastSent = now()

- Refer back to Figure 5 on page 12 for the definition of the alert_state table in the events database
- This line is effectively preventing any further command actions by this command on this event

- There is also a SELECT on the history database

If a “goodnews” is then generated, it should clear all of the badnews events and run the script for the Clear Command.

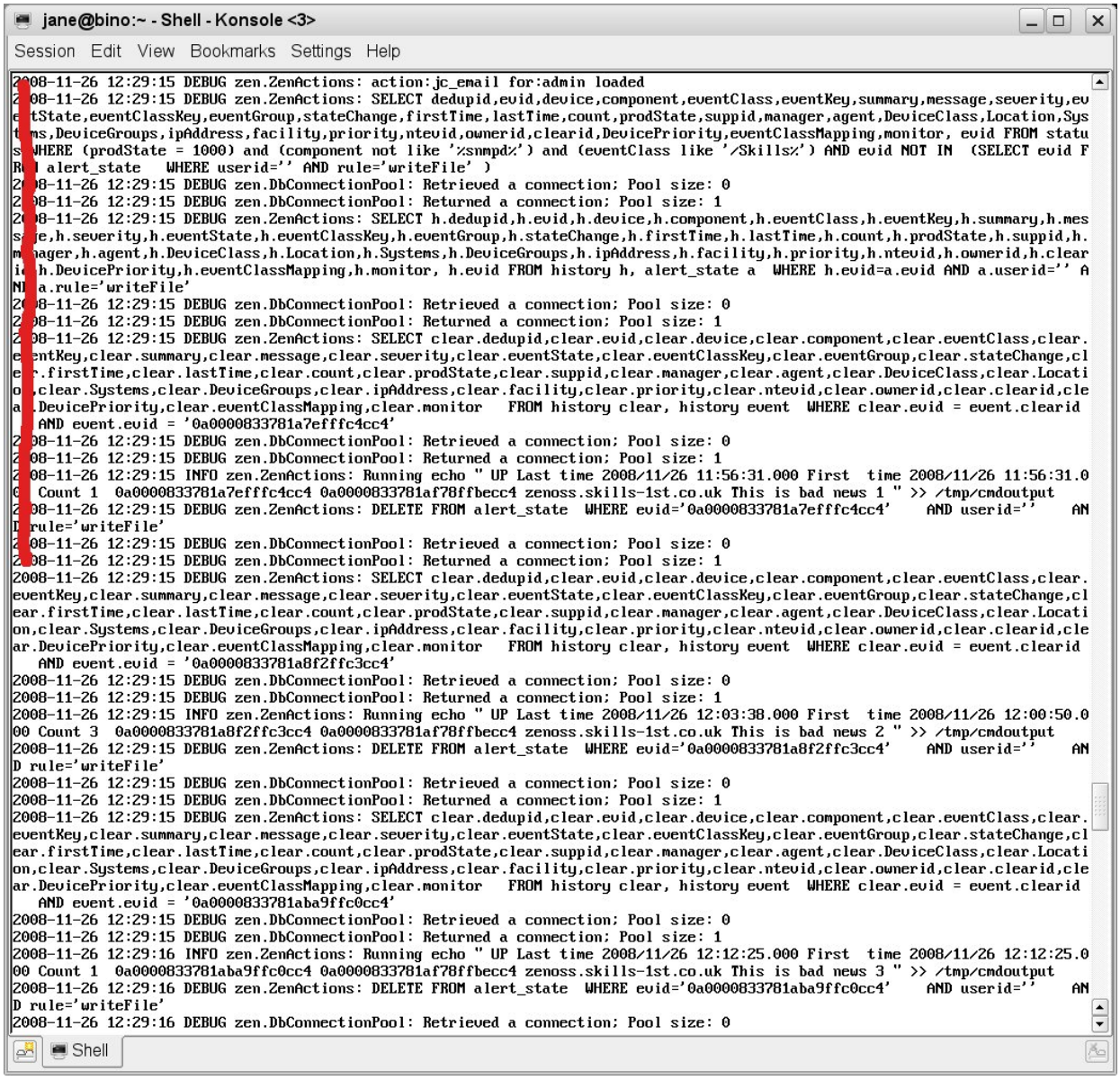

Figure 42: zenactions.log showing clearing event commands

zenactions.log should show:

- SELECT statements retrieving data from status and history databases

- An echo command for each cleared event
- A “DELETE FROM alert-state” statement for each cleared event ,which is effectively clearing the duplicates flag for the writeFile command for each event

Interestingly, inspecting the output file generated by the event command, shows that the event *firstTime*, *lastTime* and *count* fields **are** available to event commands (unlike during the event mapping process), since the “goodnews” lines that are output include the correct values for the duplicated event. This is perfectly reasonable as *zenactions.log* is showing that the information is actually queried from the MySQL status and history database tables.

10 Events, Alerts & Production Status

10.1 Alerting rules for email and paging

Zenoss provides two standard mechanisms for reporting events to users – **email** and **paging**. These are setup on a per-user basis. The mailserver / pageserver host parameters need to be defined once for a Zenoss system. Use the left-hand *Settings* menu to configure parameters for a mail server.

Figure 43: Setting up a mailserver destination for the Zenoss system

Note that this configuration is generic for the whole Zenoss system,for **sending** emails; individual users also need configuration to **receive** emails or page alerts. User parameters can either be configured from the left-hand *Settings* menu, and select the *Users* tab; or, use the *Preferences* link at the top-right of the Zenoss GUI.

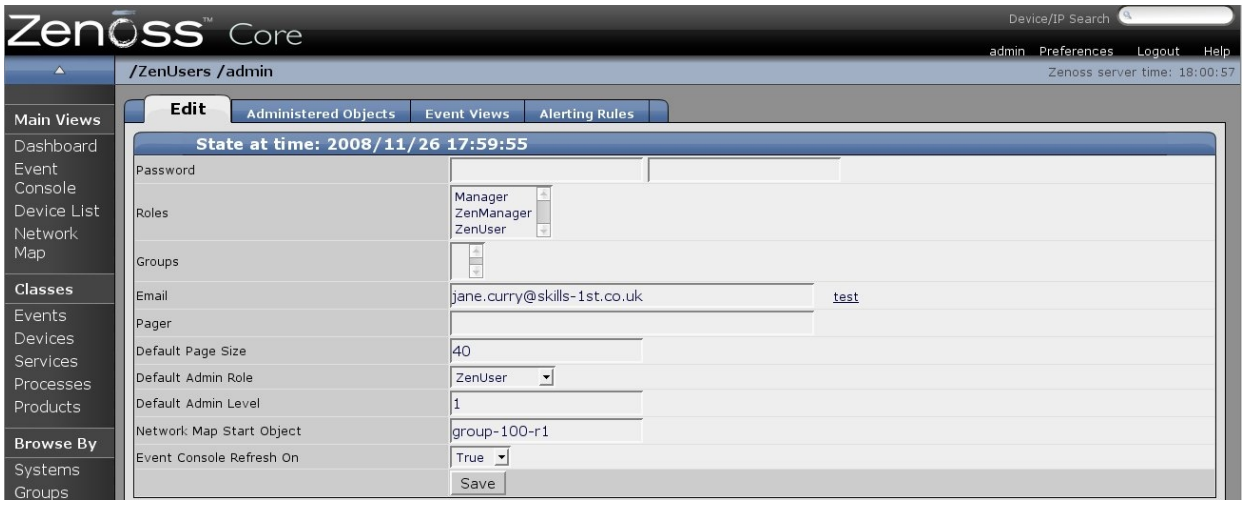

Figure 44: Configuring email destination address for each user

Note that there is a handy *Test* button alongside the email address.

Alerting rules are also configured on a per-user basis to determine **what** events to alert on, **how** to alert and **when** to alert. The alerting rules use the same filtering mechanism as Event Commands to decide which events should generate alerts.

To setup alerting rules, navigate to a user's *Preferences* page and select the *Alerting Rules* tab. Use the dropdown table menu to *Add Alerting Rules* and provide a name.

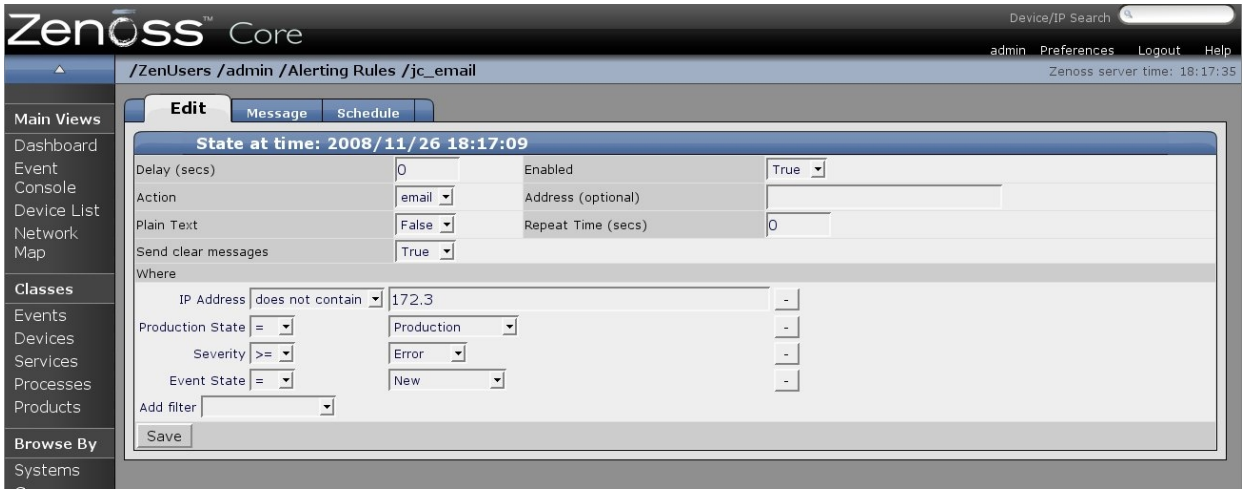

Figure 45: Defining method and filters for Alerting Rules

Note that if the email address field is left blank, the address configured in the user's Preferences page will be used.

Emails can be delayed for a period after an event has occurred and the alert can be repeated, if required.

The bottom half of the Alerting Rules *Edit* panel defines **what** events generate alerts; it is very similar to the filters seen in Event Commands. A combination of filters can be applied to determine whether an Alert is sent, or not. As with Event Commands, filters are logically ANDed.

The *Message* tab defines the content of the email to be sent. The message format is a Python format string with useful advice at the bottom of the panel as to what fields are available for substitution. The screenshot below is the default.

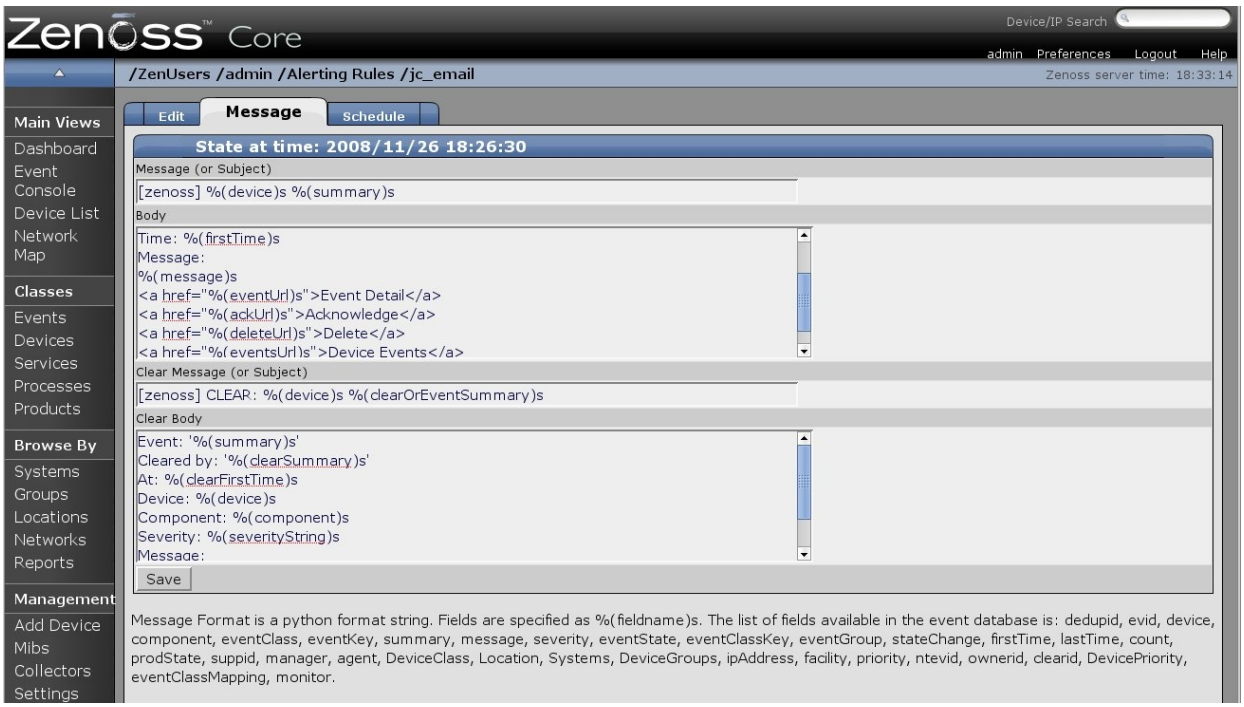

Figure 46: The message configuration of an email alert

Note that in the above screenshot, the *Body* message has further lines above that detail the Device, Component and Severity, similar to those lines seen in the *Clear Body* panel.

The *Schedule* tab can be used to restrict when alerts should be active. By default, they are always active.

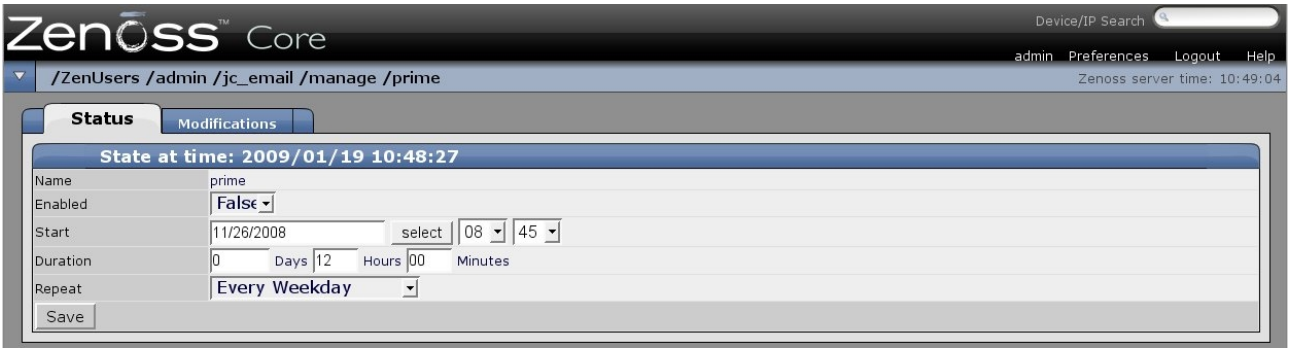

Figure 47: Alerting rule schedule example

Alerts are generated by the same **zenactions** daemon that runs Event Commands.

10.2 Other alerting possibilities

Although Zenoss only provides for email and paging out-of-the-box, it is perfectly possible to craft other alerting mechanisms using Event Commands. For example, an Event Command could be used to drive an SNMP TRAP script, passing parameters from

the Zenoss event in TRAP varbinds, to a higher-level SNMP manager. Here is a simple TRAP generation example:

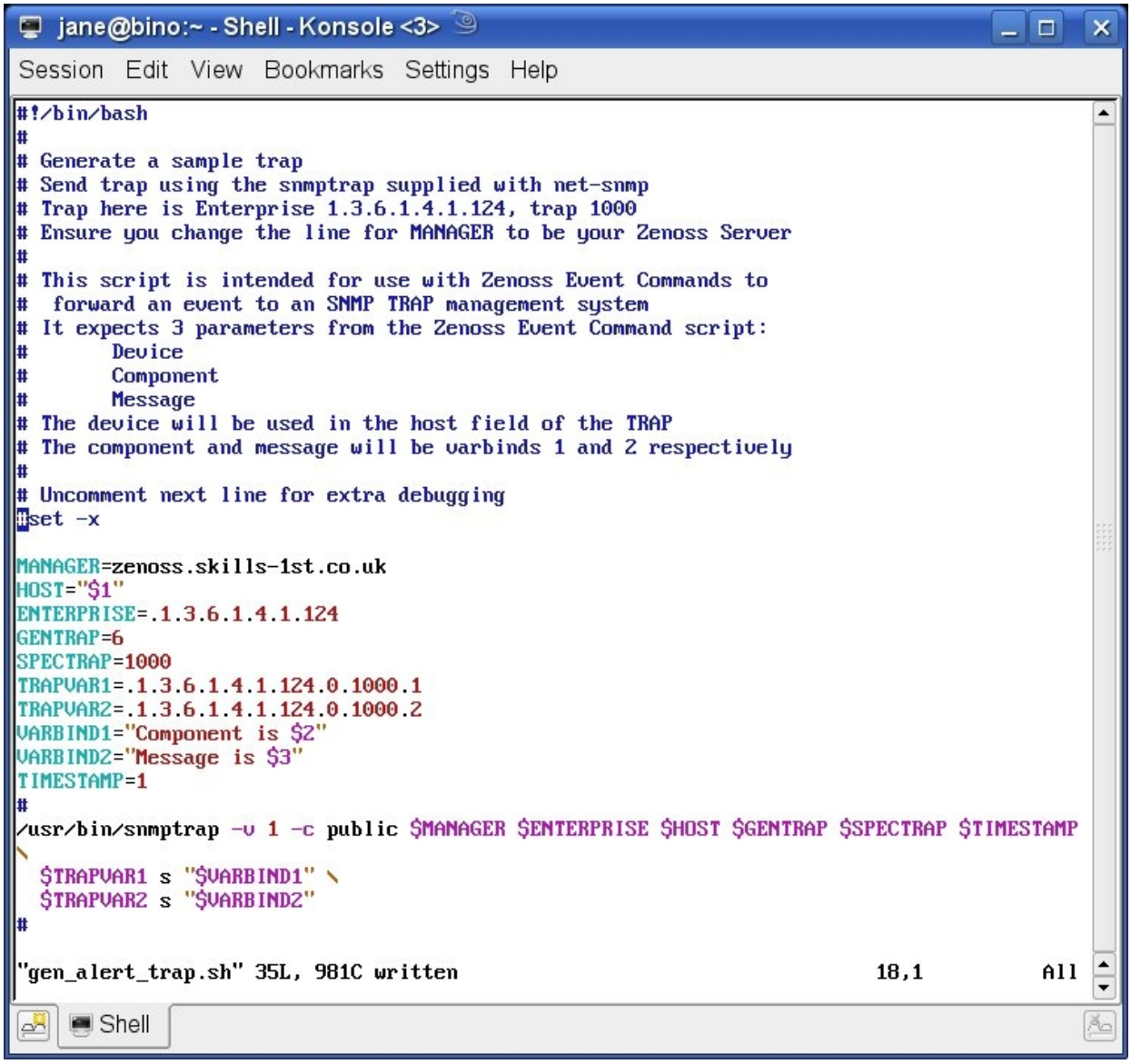

Figure 48: gen_alert_trap script to generate SNMP TRAP with event parameters

The script is then driven by a Zenoss Event Command:

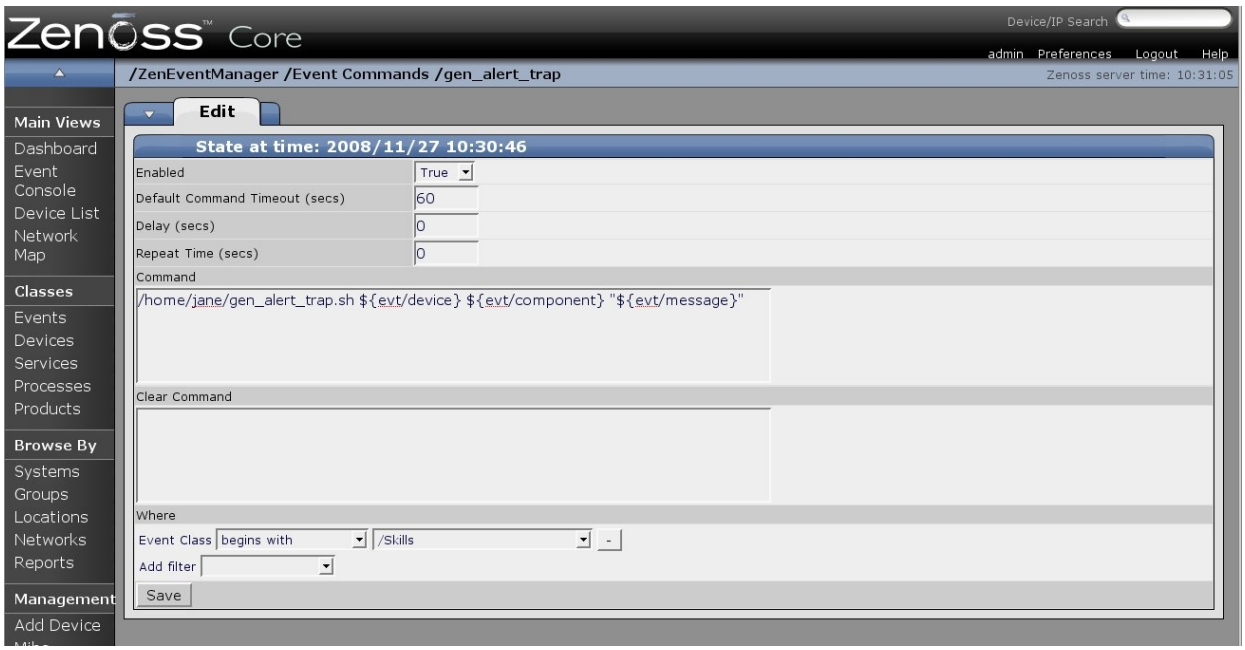

Figure 49: Zenoss Event Command to drive SNMP TRAP script

10.3 The effect of device Production Status

The **Production Status** of a device can be used to control different management aspects of a system. Production Status for a device is configured in the *Edit* tab of a device's home page.

Chapter 8 of the Version 2.4 Zenoss Administration Guide describes the different Production States and the effect that these have. Three different types of "management" are defined:

- Monitoring — ping polling and event generation
- Alerting — generating alerts (emails, pagers, event commands)
- Dashboard — whether to include in the *Device Issues* portlet

In practise, anything to do with alerting, including event commands, is controlled by the filters in the alerting rule or the event command. If no *Production State* filter is configured in the alerting rule or event command, then the alert / command **will** run, by default.

A device Production Status of *Production* will result in events contributing to the *Device Issues* portlet of the Zenoss Dashboard and in IP monitoring taking place. Alerts will be generated and event commands will be executed if their filters permit.

A production Status of *Decommissioned* should result in IP monitoring ceasing; hence, all events will cease and no alerts will be generated. The device will not be recorded in the Dashboard *Device Issues* portlet. **Note** that the overall Status icon on a device's Status page will turn **green** ! If the device is already down when the status is set to *Decommissioned* then you should see existing "Interface down" messages cleared to History by a clearing event whose message is "No longer testing device ...".

Any Production Status **other** than *Production* will result in the device **not** being included on the Dashboard *Device Issues* portlet.

The Production State of a device can be changed automatically to allow for maintenance windows. This is achieved from a device's status page – select *More* from the table dropdown menu and *Administration*. Maintenance windows are displayed; they can be modified, added and deleted.

The example below shows the maintenance window for a backup system that only runs for about 3 hours at night, between 23:45 and 3:00. From 3:00, for 20 hours and 50 minutes, it is moved to a Production Status of *Original* , which for this machine is defined as *Decommissioned* . Thus monitoring, events and alerts are only generated during the few hours that this backup system is up.

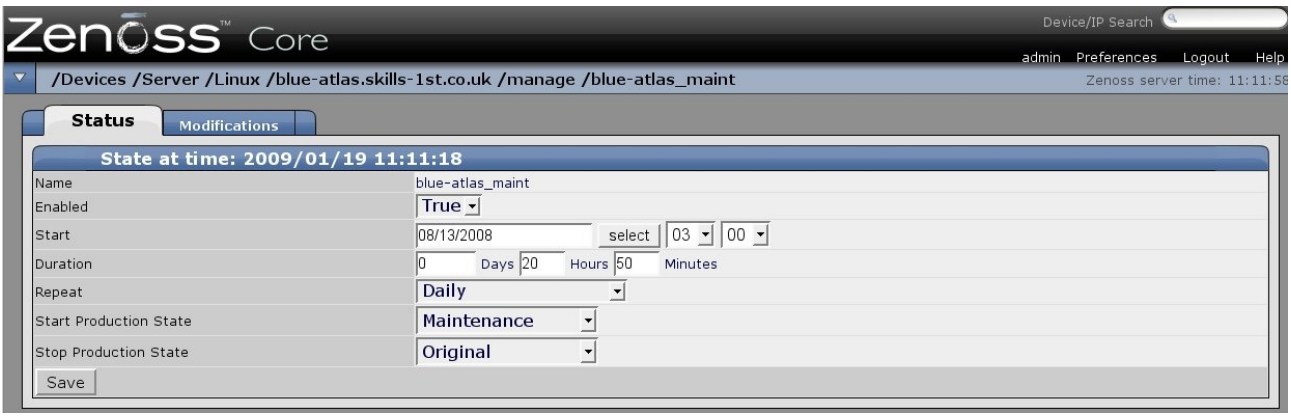

Figure 50: Maintenance schedule for machine that only runs from 23:45 until 3:00

Note that maintenance schedules can also be applied to device class hierarchies; they do not have to be applied to specific devices.

11 Conclusions

Zenoss has an extensive event system capable of receiving events from Windows, syslogs and SNMP TRAPs, in addition to receiving the events generated internally by Zenoss's own discovery, availability and performance monitoring.

A large number of event classes are defined and configured when Zenoss is installed. These can be modified, removed or added to.

An event follows a fairly complex event life cycle process whereby it is mapped to an event class and then, optionally, it is transformed such that default fields of the event can be changed and user-defined fields can be created.

Event mapping for events from Windows, syslogs or SNMP, depends on the initial Zenoss parsing daemon delivering an eventClassKey field which must correspond to a defined mapping. Subsequently, a Python Rule and/or a Python Regex can be used to further distinguish between incoming events and map to different event classes.

An event class includes event context – zEventAction, zEventSeverity and zEventClearClasses – which can be applied to individual subclasses of events or to class hierarchies. This means transforms can be affected by event type.

Device context is also applied to an incoming event; device context includes the prodState, Location, DeviceClass, DeviceGroups and Systems field values. Device context provides the ability for transforms to take account of the device or device class hierarchy.

Event transforms can be simple assignment of event fields or can include complex Python programs. A good environment for testing Python is the zendmd command line utility. Transforms and/or the event context can be used to help clear events that have been resolved. Any event with a severity of Cleared will automatically clear other similar events; zEventClearClasses can be used to list extra classes that are cleared in addition.

Events are saved in the MySQL events database. By default, events go to the status table; when they are cleared, they are moved to the history table.

When events occur, alerts can be generated using email or a paging system; alternatively, any script can be run on the Zenoss system, as an event command.

As with any enterprise management system, Zenoss has the tools to configure almost any response to any event.

12 Appendix A zendmd commands useful with events

zendmd is a utility supplied by Zenoss which provides a Python shell environment with access to the Zenoss object database (ZEO). This can be useful, especially for testing event mapping transforms.

zendmd should be run as the Zenoss user. Command recall (up-arrow key) is available and extremely useful.

Note that Python is very particular about line indentation; some syntax requires extra indentation (for example the body of a *for* loop). It does not matter how many spaces are used but the number **must** be consistent for the whole of that body.

- Print all event class mapping instances for an event class, *Skills :*

```
print dmd.Events.Skills.instances.objectIds()
```

- Print all events with their event id:

```
print dmd.ZenEventManager.getEventList()
```

- Print all event classes:

```
for ec in dmd.Events.getSubOrganizers():
     print ec.getOrganizerName()
```

- Print all event classes (not event class mappings) that have transforms:

```
for ec in dmd.Events.getSubOrganizers():
      if ec.transform:
            print ec.getOrganizerName()
```

- Setup the variable *evt* to point to an existing event. This is an extremely useful testing technique!
  - From an Event Console, bring up details for an event. Copy the event id.

```
evt=dmd.ZenEventManager.getEventDetailFromStatusOrHistory("<paste id>")
print evt.summary
print evt._details
```

- Print all attribute / value pairs for an event. This includes user-defined event fields in the *_details* field and also the fields for event zProperties (again with attribute names prefaced with an underscore (eg. *_action*). Note that this command does **not** show methods for evt, only attributes. Note the syntax around "dict" below, is 2 underscores before and after. Also note the line has 2 dots in it and a colon on the end!

```
for key,value in evt.__dict__.items():
      print key,value
```

- Print attributes **and** methods for an event. Note that the "x=" line could be omitted and the print line have "x" substituted by *getattr(evt,attr)*

```
for attr in dir(evt):
     x=getattr(evt,attr)
     print attr,x
```

- Print a list of the data for an event, the details for an event and the fields of an event:

```
print evt.getEventData()
print evt.getEventDetails()
print evt.getEventFields()
```

- Two different ways to get attributes from an event. The latter will return a null string if the attribute is missing:

```
evt.<attribute>
getattr(evt, <attribute>, '')                Note – 2 single quotes before
)
```

- If things get horribly messed up, try:

```
dmd.Events.reIndex()
commit()
```

References

1. Zenoss network, systems and application monitoring - http://www.zenoss.com/
2. Zenoss Administration Guide http://www.zenoss.com/community/docs
3. Zenoss Developer's Guide http://www.zenoss.com/community/docs
4. Zenoss Frequently Asked questions (FAQs) - http://www.zenoss.com/community/docs/faqs/faq-english/
5. Zenoss HowTos - http://www.zenoss.com/community/docs/howtos
6. Zenoss wiki - http://www.zenoss.com/community/wiki
7. For documentation on Zenoss functionality, the ZEO object database and Zope. try http://www.zenoss.com/community/docs/zenoss-api-docs/2.1/
8. "Zenoss Core Network and System Monitoring" by Michael Badger, published by PACKT Publishing, June 2008, ISBN 978-1-847194-28-2 .
9. SNMP Requests For Comment (RFCs) - http://www.ietf.org/rfc.html
10. V1 – RFCs 1155, 1157, 1212, 1213, 1215
11. V2 – RFCs 2578, 2579, 2580, 3416, 3417, 3418
12. V3 – RFCs 2578-2580, 3416-18, 3411, 3412, 3413, 3414, 3415
13. As a general Python reference, try "Learning Python" by Mark Lutz, published by O'Reilly
14. SNMP Host Resources MIB, RFC s 1514 and 2790 - http://www.ietf.org/rfc.html
15. For information on TALES expressions, see http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/AppendixC.stx
16. For information on Python regular expressions, see http://www.python.org/doc/2.5.2/lib/re-syntax.html and http://docs.python.org/dev/howto/regex.html
17. For the extension SNMP MIB from Informant, go to http://www.wtcs.org/informant/index.htm
18. Good sites to look for Cisco MIBs and their prerequisites are:
    - http://tools.cisco.com/Support/SNMP/
    - ftp://ftp-sj.cisco.com/pub/mibs/ including V1 versions of V2 SNMP MIBs
19. Zenoss ZenPack site for the MIB Browser ZenPack - http://www.zenoss.com/community/projects/zenpacks/mib-browser
20. Datagram Syslog Client http://syslogserver.com for syslog Windows systems.
21. Raddle network emulation open source package - http://raddle.sourceforge.net/
22. "Zenoss Event Management Workshop" available from Skills 1st Ltd, http://www.skills-1st.co.uk/products/courses/

About the author

Jane Curry has been a network and systems management technical consultant and trainer for 25 years. During her 11 years working for IBM she fulfilled both pre-sales and consultancy roles spanning the full range of IBM's SystemView products prior to 1996 and then, when IBM bought Tivoli, she specialised in the systems management products of Distributed Monitoring & IBM Tivoli Monitoring (ITM), the network management product, Tivoli NetView and the problem management product Tivoli Enterprise Console (TEC). All are based around the Tivoli Framework architecture.

Since 1997 Jane has been an independent businesswoman working with many companies, both large and small, commercial and public sector, delivering Tivoli consultancy and training. Over the last 5 years her work has been more involved with Open Source offerings.