Eddie (Essential Distributed Diagnostic and Information Engine)

Rod Telford (rtelford@connect.com.au)
Chris Miles (cmiles@connect.com.au)

Monitoring in an environment the size of CONNECT's has presented us with some interesting challenges. This paper describes a tool we have developed to help us overcome these challenges.

Introduction

Most system administrators have a collection of scripts, written over time, to perform various monitoring tasks; a sub-optimal solution at best. Unsatisfied with available solutions, we set about developing a tool to meet the monitoring demands of a system the size and complexity of Connect's. Eddie is the result. Eddie is a highly configurable and easily extendible system monitoring tool, written entirely in Python. It is therefore fully portable to a wide variety of operating systems.

This paper examines the challenges system administrators face in effectively monitoring the health of their systems, explores currently available freeware monitoring systems, and discusses the shortcomings of these tools. Eddie is then introduced, explaining what advantages it has over other monitoring tools, and discussing the set of features that it provides.

Detailed information on how to easily configure and extend Eddie is included, and we conclude with a discussion on the future of this flexible administration tool. The appendix contains a porting guide to aid system administrators in deploying Eddie to other platforms. The appendix also contains an agent configuration tutorial to help you get started with Eddie.

A Brief Survey of Available Tools

We have had a good look around the Internet for freely available system monitoring tools. We discovered a variety of software which, despite having many good features, fell short of our requirements. The following is a summary of the pros and cons of the software we examined.

Sysmon

ftp://puck.nether.net/pub/jared/

version tested: 0.78.3.2

Sysmon is a centralized monitoring server that can be configured to know about network dependencies. It comes with a simple curses-based client which connects to the Sysmon server and displays information of the systems being monitored. Updates are instantaneous. The server also produces a web page listing any systems which have problems so the user can check on the status of their systems via a web browser.

Sysmon can monitor hosts and network devices, and can check numerous protocols/ services including ftp, http, smtp, pop3, imap, telnet, x.500.

It is written in C and the client and server must be compiled before use.

Advantages:

- Network dependency tree
- Numerous protocol checks
- Terminal (curses)-based or web-based clients
- Configuration file is fairly easy to understand.

Disadvantages:

- Centralized server monitors all protocols, allows for no redundancy
- Written in C makes porting and adding new features and protocols difficult
- Notification appears to be by email only
- No support for intelligent notification or escalation

The Big Brother System & Network Monitor

http://maclawran.ca/bb-dnld/

ver 1.07a

Big Brother is a shell-script driven system which can check the status of various common protocols. It provides notification via email, paging and SMS as well as a status display in a web page.

Is mostly written in shell-script, with a few compiled utilities

Advantages:

- Most common protocols can be checked
- Web display provides easy to read status of hosts and protocols
- Grouping of hosts in config is handy

- Supports paging and SMS as well as email notification

Disadvantages:

- Mixture of shell-scripts and C programs means maintenance and addition of new features is not trivial
- Configuration file layout is not very intuitive
- Web display did not provide as much detail as we would like
- No support for intelligent notification or escalation

Patrol

version tested: 2.0

Patrol is a simple monitoring program written in Perl. It provides monitoring of services and state of the local host it is running on, and can check the status of file-systems, processes, pid-files and service ports.

Advantages:

- Config file is easy to read and maintain
- Message definitions can contain variables which are substituted for when the message is sent.

Disadvantages:

- Code is not well-written making it difficult to maintain and add new features, plus we found quite a few bugs during use;
- Monitoring is limited to services running on the local host;
- No support for intelligent notification or escalation.

Our Findings

We have found that Patrol had the closest feature set to what we require. However Patrol was not particularly easy to expand or maintain as the code base was extremely ugly. We set about designing a system that would solve all CONNECT's system monitoring needs, and meet our personal goals and expectations for such a system.

Introducing Eddie

What is Eddie?

Eddie is a platform independent distributed monitoring utility, designed to meet the need of large networks and complex systems. Eddie has been under constant development for the last 12 months, starting as a simple replacement for Patrol and evolving into the feature rich application presented in this paper.

Eddie Design Considerations

What Were Our Goals When We Were Designing Eddie?

When designing Eddie our goals were:

1. Platform independence - We run Solaris 2.51, Solaris 2.6 for Intel and Linux
2. Powerful but friendly configuration
3. Intelligent notification
4. Separation of the notification system from the monitoring system
5. Reliable monitoring and notification - This is a monitoring system after all :-)
6. Distributed monitoring - we need to monitor multiple systems spread over the country.

How Did We Achieve These Goals?

1. *Platform independence*

To achieve the level of platform independence we wanted, we chose to use a cross-platform programming language called Python. Python is available for most modern Operating Systems, including all flavours of UNIX and Windows NT. Our goal was to make the core of Eddie platform independent and have platform specific modules separated from the core so that supporting a new platform was as simple as porting the platform specific modules.

Other important reasons for choosing Python over other platform independent languages (such as Perl or Java) was speed of development, and its clean object-oriented nature which allowed us to build Eddie out of objects which could be plugged together to form a complete monitoring solution.

2. *Powerful but friendly configuration*

We attempted to strike a balance between ease of configuration and flexibility. We decided to follow Python's object-oriented model by designing the layout of the configuration files to be almost identical to the layout of Python code. That is, indented by tabs, with 'configuration objects' created by a keyword followed by a name followed by a colon, then the object's properties (which may also be 'configuration objects') indented below the parent object. [see Appendix A]

We have found this configuration layout to be much simpler and more logical for the object-based configuration mechanism that we have designed. It is also very logical for users who are already familiar with Python programming or object oriented techniques.

3. *Intelligent notification*

One of our major goals in designing a new monitoring system was to build in some flexibility for handling alert notifications, we discuss this in some detail because it is one of the most important aspects of system monitoring. In the past we suffered from monitoring packages which would, for example, page us every 10 minutes to remind us that a disk was still full (doh!). What we needed was a way to acknowledge system alerts, and give Eddie the ability to track problem status. This eliminates the problem of receiving the same report every ten minutes. (Of course Eddie will still send out the occasional reminder)

Another ‘smart monitoring’ feature we wanted was the ability to define dependencies, so that we would not be unnecessarily notified about problems which were caused by larger, more obvious problems. For example, we don’t care to be notified that the NNTP port is not responding if innd is not actually running. And we don’t care if innd is not running if the box is not responding, etc.

4. *Separation of the notification system from the monitoring system*

As we were designing the project we found that it was not necessary for Eddie to be totally responsible for the messaging system. We found that there were already several powerful messaging systems in development which we could take advantage of, such as Elvin [SEGALL 97]. After experimenting with Elvin we decided that we would design a simple messaging system for Eddie, but also allow easy integration with other proven messaging systems like Elvin. Initially a user could set up a simple system with built-in Eddie notification, then at a later date move to a more advanced, redundant, messaging environment by plugging it into a system such as Elvin.

5. *Reliable monitoring and notification*

After having problems with other monitoring systems failing to relay notification messages to us in an accurate and timely manner, we knew that a major Eddie design goal would be to provide redundant and reliable monitoring, along with reliable messaging.

We can achieve reliability by having Eddie check that it is running on neighbouring hosts and reporting any sign of Eddie behaving incorrectly. Redundant monitoring would be achieved by having multiple neighbour hosts checking each other.

Reliable messaging can be achieved by the messaging system (Eddie, Elvin or otherwise) ensuring its messages are delivered successfully. If the notification method fails (or if no acknowledgement is received within a certain time) then the messaging system can fallback to different notification methods, until the message is delivered successfully.

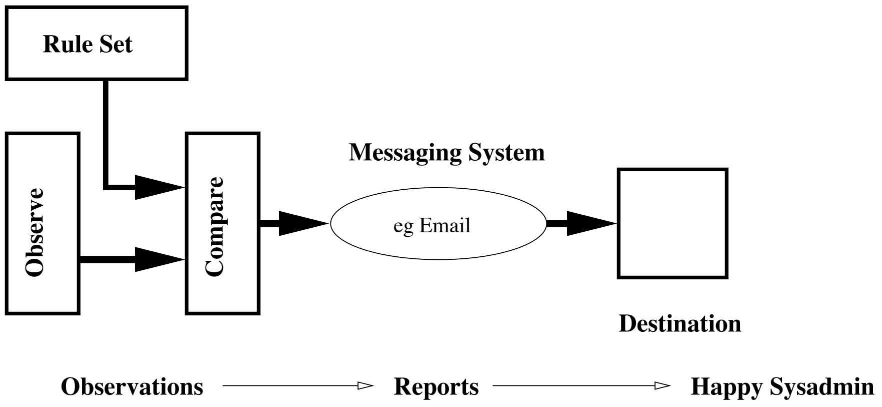
6. *Distributed monitoring*

Finally, an important design goal was to provide a distributed monitoring environment. This would be best achieved by giving Eddie the intelligence to know about network structure. So, along with monitoring services on the local system, Eddie would also monitor neighbouring systems, network devices and networks. We plan to implement this in such a way that multiple notification messages would not be delivered for the same problem.

The Generic Eddie Architecture

Description

Because of our design objectives, our architecture for Eddie needed to be simple but also able to accommodate distributed monitoring and messaging. We settled on the following generic structure:



Implementation

We concentrated on writing the data collection components of Eddie first. This allowed us to obtain information for testing, and to help refine interfaces to the other Eddie modules.

Our initial configuration file was based on the Patrol configuration file format. After trying to extend this format, we decided that it would be more appropriate to design a configuration system which better suited the object oriented nature of Eddie.

We implemented rules as Python objects - this neatly reduced Data Processing to a series of method calls. The Messaging framework was only slightly more complex (although implementing SMS paging proved to be an interesting challenge).

Deployment Considerations

Eddie can be deployed in a variety of ways, from simple to complex. In this section we describe only the most basic deployment architecture and introduce our advanced concepts after discussing some necessary background information. When we refer to complexity in this paper, we are talking about the amount of time and effort needed to deploy Eddie. It is useful to divide this complexity into 4 levels. From simplest to most complex these level are:

1. Local Agent
2. Distributed Reporting
3. Distributed Observations
4. Distributed Everything

To fully explain how these levels are deployed, we need to divide the monitoring process into three distinct phases;

- **Data collection** - This is the observation phase. We collect output from various programs, such as, df, netstat, uptime and ps. (In the future, we plan to collect some of this information directly from the kernel.)

- **Collation** - The observations are then compared with known rules, which are defined in Eddie's configuration files.
- **Reporting** - At this stage we are concerned with getting any information we have gathered to the intended destination.

Local Agent

We designed Eddie to be extensible, but also wanted it to be useful in simple configurations. The most basic Eddie configuration is the Local Agent.You simply install and configure the Eddie agent on the machine you wish to monitor. At this level, all three stages of the monitoring process are handled by the Local Agent. The agent collects data, compares the data with rules that have been written into your custom configuration file, then reports (usually via email) about any problems it has found. This method should be satisfactory for most installations, however you then have a reliance on your email system to deliver the message. Most people would not consider this to be a major issue, but email is a store and forward protocol and there is no guarantee of timely delivery of your problem reports.

Configuring the Local Agent

Configuring the Local Agent is as simple as creating your configuration file to monitor the things of interest to you, Eddie currently supports the following directives.

- COM - allows a command to be executed and the output and/or return value of the command can be compared against expected values or strings;
- PROC - checks for processes either running or not running;
- FS - check file system usage (or space remaining), either instantaneous or averaged over a period of time;
- PID - check for existence of pid-file and/or check that pid inside pid-file is actually a currently running process;
- SP - check that a service port is currently active and is listening for connections;
- URL - check URL is currently active;
- PING - check host is returning pings.

Please refer to Appendix A for a detailed configuration example.

E-Mail vs Messaging

Many System Administrators would prefer a more reliable mechanism than email to alert them about system problems. After all, how will you be alerted if the email system is down?

We believe that delivering a message, in a timely manner, to its intended destination is one of the most important aspects of system monitoring.

So how do we ensure our messages reach their intended destination in a timely and reliable fashion? We need to introduce a messaging federation and move to distributed reporting.

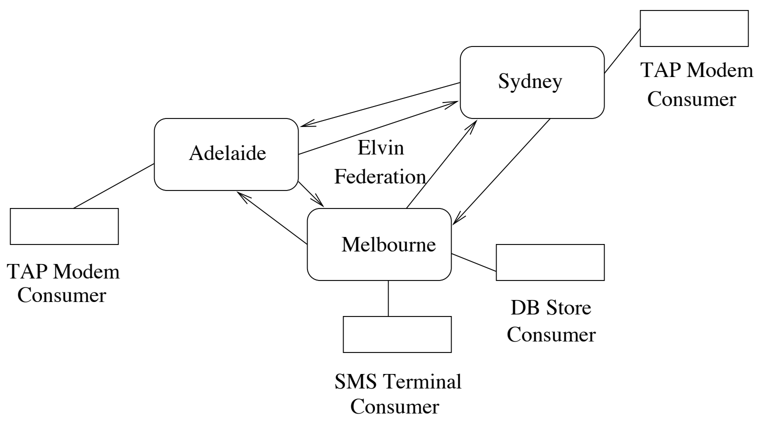
Who is Elvin Anyhow?

Before we move on to discuss advanced Eddie deployment, let me introduce Elvin. Quoting from the Elvin home page[1]:

"Elvin is a publish-subscribe notification service. Its combination of features make it unique amongst notification services. Elvin delivers unaddressed notifications which are received by consumers on the basis of the notification's content. Consumers subscribe to announcements satisfying some constraints over their contents."

I discovered Elvin on a recent visit to the DSTC, after a long chat with Bill Segal, one of Elvin's authors, I was convinced Elvin was the right tool for Eddie. Elvin has a great feature set, which includes a Python API which just happened to accept notifications identical to the way we were generating them.

One of the major motivations for using Elvin, is its ability to be configured into a *federation*. A federation is a collection of interacting Elvin servers that share the notifications produced by Eddie. The federation should also share information on the availability of resources such as, SMS modules, and modems for TAP[2], Eddie should be configured to do this as part of your federation configuration. A federation may look like:



A full discussion on how a setup a federation is well beyond the scope of this document, however we plan to write a guide on configuring Elvin for Eddie. Supporting modules are already provided.

Advanced Eddie Deployment

Recall that our four levels of architectural complexity are:

1. Local Agent (Described Above in "Generic Eddie Architecture")
2. Distributed Reporting

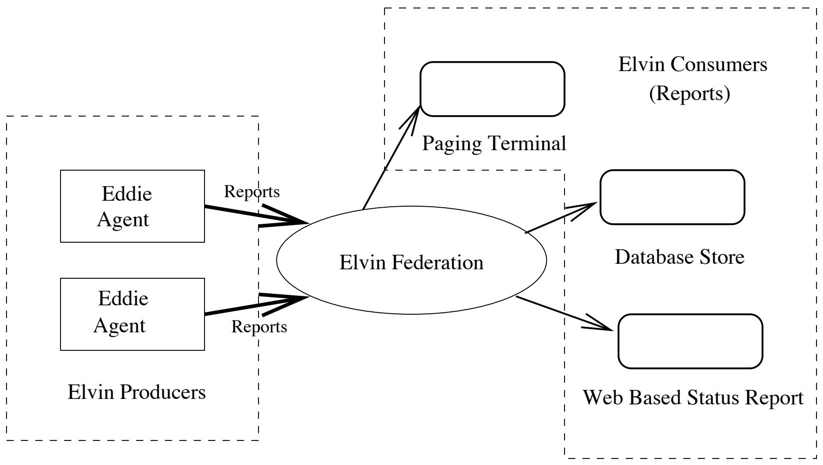
1. http://www.dstc.edu.au/Elvin
2. TAP is a protocol used for talking to radio paging providers

3. Distributed Observations
4. Distributed Everything

We have already seen Local Agent and will now introduce the other three.

Distributed Reporting

With distributed reporting we pass the responsibility of reporting to a messaging federation. Eddie currently supports Elvin, however there is no reason the Elvin module could not be replaced with support for other messaging systems. In this setup, the Eddie agent collects and collates the data, then passes the reports to Elvin. Once the reports are published it is then up to an Elvin consumer to pass the report on to intended destination.



The advantages of handing off reporting to other systems are numerous, and include;

- **Redundancy** - Elvin can be configured into a messaging federation, which provide multiple message paths to the message recipients. (see section “Who is Elvin Anyhow?”)
- **Distribution of information** - Information published into the federation is easily accessed by many other systems. This allows you to write utilities to retrieve the reports and store them in a database. You could provide information streams to pretty front end web pages, or almost anything else you can think of.

This configuration bypasses the use of email, thus eliminating the store and forward problem. Many more reporting options also become available because the data is more accessible. For example we have a Seimens M1 GSM module[1] which transmits SMS messages straight onto the Vodafone network. This support is in the Eddie contrib area as are many other Elvin consumers for Eddie. See Appendix C for a practical Distributed Reporting example.

Distributed Observations

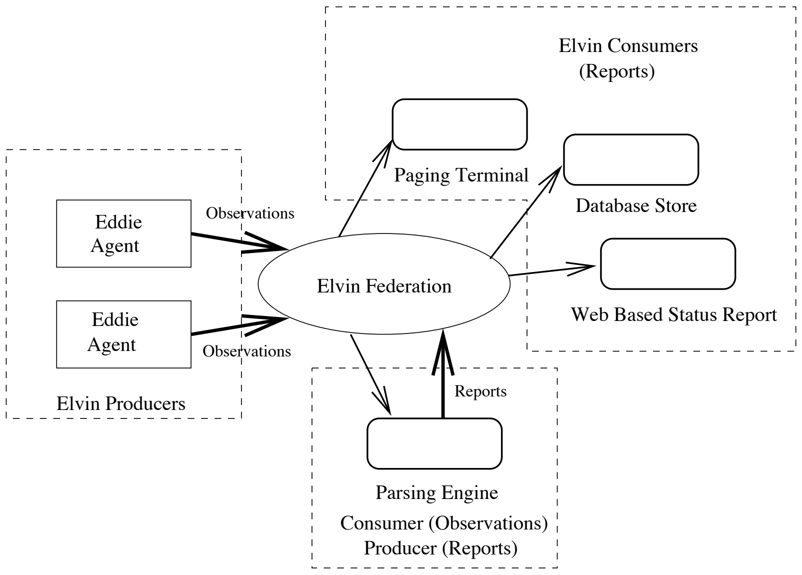
Distributed Observations make use of the Elvin federation at a much higher level. At this level we pass off our observation to the federation. Elvin consumers are then responsible for parsing the data and generating higher order Elvin notifications, sim-

1. Refer to http://www.uq.net.au/pacificdata/siemens/gsm.htm for information on the Seimens M1 GSM module

ilar to the notifications generated by the Eddie Agent when using Distributed Reporting.

In addition to the benefits of the Distributed Reporting configuration, Distributed Observations shares the data observed by the agent, which provides the following benefits:

- **Current Statistics** - At any time you can see the last observation for any statistic from any host.
- **Observations over time** - you can store your observation in a database and collate data over long periods of time. This might also be useful when analysing a break-in or system crash, or for diagnosis of problems that are only apparent over time, such as gradual performance degradation.
- **More Information** - This setup produces a much richer information base for any systems examining the data. This allows you to make statements on the behaviour of your system as a whole, which could be useful when preparing reports, or when justifying new hardware to your management.



This configuration simplifies the Eddie agent setup, as you can run the same agent configuration on all systems. The major task is setting up the Elvin consumer to generate higher order notifications.

Distributed Everything

Complexity level four is theoretical and involves the setup of a parallel virtual machine (PVM)[1]. This allows us to treat all our systems as one PVM running one application, Eddie. This is a useful abstraction but is beyond the scope of this docu-

1. Refer to http://www.epm.ornl.gov/pvm/ for details

ment. However, we would be happy to muse over the infinite possibilities this concept may yield with anyone who is interested

TABLE 1.

Complexity	Eddie Agent	Federation
Local Agent	Observation Collation Reporting	
Distributed Reporting	Observation Collation	Reporting
Distributed Observations	Observation	Collation Reporting
Distributed Everything *(theoretical)*		Observation Collation Reporting

The table above summarises the various complexity levels and show the function of the Agent and Federation at each level.

Summary

Upon undertaking this project our goals were to build a system that could meet the monitoring requirements of a system like Connect's, and also to write a quality application that was both extensible and maintainable. We believe that these goals have been met and Eddie has become an essential part of our sysadmin tool kit at Connect. However, Eddie is a constantly evolving package, we are always looking to improve the design, and add new features to the software.

Eddie is available from the SAGE-AU ftp site, and I encourage you to download it and give it a go. We would also enjoy any suggestions or modifications you would like to contribute.

References

[Segall97] Elvin has left the building, Segall and Arnold, 1997.

[RFC 1861] Simple Network Paging Protocol, A Gwinn, October 1995

Acknowledgments

Thanks to Mike Ciavarella for his assistance in putting this paper together.

Appendix A - Agent Configuration Manual

How do you configure Eddie?

Eddie configuration was designed to be object-oriented in nature and the layout was borrowed from Python code layout. That is, object properties are indented from the object parent which makes for a neat and easy to read configuration format (in our opinion).

Eddie configuration usually begins with a default config file as shown below.

```
   # Eddie Global Config File
   # SAMPLE
   #

1  LOGLEVEL=8                    # log levels: see doc/config
2  ADMINLEVEL=5                  # Admin inform level
3  LOGFILE="/var/log/eddie.log"    # logging details go into this folder
4  ADMIN="cmiles@connect.com.au"   # The Eddie Admin
5  ADMIN_NOTIFY=1d               # Send Admin summaries only once a day

6  SCANPERIOD=10m                # scan every 10 minutes

7  INTERPRETERS='sh,bash,perl,perl5.001,perl5.003,perl5.004,python,python1.4,python1.5'

8  DEF SYSPAGER=123456

9  INCLUDE 'news.rules'
   INCLUDE 'vws.rules'
   INCLUDE 'proxy.rules'
```

We will explain the important points of the config file as follows:

1. The LOGLEVEL directive specifies the amount of information that will be written to the Eddie log file. The log levels are defined as follows:
- 0 - no logging at all
- 1 - log serious problems - problems which may crash program
- 2 - log important errors - which may cause incorrect operation
- 3 - log alerts - do not affect program operation, but aren’t good
- 4 - log warnings - don’t affect anything but should be noted
- 5 - log actions taken
- 6 - log informative messages
- 7 - log user-debug messages - usually all checks performed
- 8 - log most things except multi-line crap - good for debugging
- 9 - log EVERYTHING! - use with caution!
2. ADMINLEVEL defines the amount of information that will be logged and then delivered to the ‘Eddie Administrator’ periodically.
3. LOGFILE defines the location of the Eddie logfile.
4. ADMIN defines the email address of the ‘Eddie Administrator’. This address is used to send administration messages (currently just log messages as defined by ADMINLEVEL).
5. ADMIN_NOTIFY defines how often administration messages should be sent to the 'Eddie Administrator'. The default time scale is in seconds, but you can specify larger time scales by following the number with one of the following characters:
- s - seconds
- m - minutes

- h - hours
- d - days
- w - weeks
- c - calendar months
- y - years

6. SCANPERIOD defines the default period between checks. This can be overridden by other directives or rule groups.
7. INTERPRETERS defines a list of standard interpreters. This is used by the process checking directive (PROC) when searching for a process name by ignoring the interpreter part of the process name. e.g.: a perl program may show up in the process list as "perl program.pl" but because Eddie knows "perl" is an interpreter, the user only needs to specify "program.pl" as the process name to check for.
8. The DEF directive allows the user to define constants for use elsewhere in the config. In this case, we have defined the constant "PAGER" and given it the value "123456". This constant can be referred to later as "$PAGER".
9. INCLUDE allows the user to split the configuration into multiple files which will be included into the current file by this directive.

An example configuration file that defines a group, notification methods, checking rules and notification messages is shown on the next page.

```
  # Eddie config file - SAMPLE
  # Config for News systems

1  group News:

    ###########################################################################
    # Notification Definitions
2    N NEWSALERT:
3         notifyperiod=10m
4         escalperiod=20m
5         Level 0:
6              email(cmiles,WARN)
         Level 1:
7              email(alert,WARN),elvin("News warning...")
              sms(sys-sms,WARN_P)
                   page(systems,WARN_P)
         Level 2:
              elvin("Alert!")
                   email(alert,ALERT)
              page(systems,ALERT_P)
                   email(alert,ALERT)
         Level 3:
              email(sysadm,"news problem")
                 email(root,ALERT),elvin("News Problem")
                 page(alert,ALERT)
                   sms(cmiles,ALERT)
              email(newsmaster,ALERT)
    ########################################################################
    # RULES

    ###PROC###
    # NR = process not running
    # R  = process is running
    #  Daemon       Rules   Action
    #  ------       -----   ------
8    PROC actived:   NR      NEWSALERT(newsmsg.proc)
    PROC innd:      NR      NEWSALERT(newsmsg.proc,1)

    ###FS###
    #  Filesystem           Thresholds      Action
    #  ----------           ----------      ----------
9    FS /opt/local/news: "capac>=90%"                        NEWSALERT(newsmsg.fs,0)
    FS /var/spool/news: "(capac>=90 & capacdelta>=2)"       NEWSALERT(newsmsg.fs,1)

    ###PID###
    # EX = check if pidfile exists
    # PR = check if process pid found in pidfile is running
    #    pidfile                   Rules   Action
    #    -------                   -----   ------
10    PID /etc/mail/sendmail.pid:     EX      NEWSALERT(newsmsg.pid,0)
    PID /etc/mail/sendmail.pid:     PR      NEWSALERT(newsmsg.pid,1)

    ###SP###
    #   Port        Bind Addr       Action
    #   ----        ---------       ------
11    SP tcp/nntp:    *               NEWSALERT(newsmsg.sp,0),depends(PROC.innd)
    SP tcp/uucp:    *               NEWSALERT(newsmsg.sp,0)

    #    command                          Rules       Action
    #    -------                          -----       ------
12    COM  "/opt/news/bin/newschk":          "ret != 0"   NEWSALERT(newsmsg.com,0)
    COM  "uptime | awk '{print $10}'":     "out > 5"    NEWSALERT(newsmsg.com,2)


    ########################################################################
    # MESSAGES

    # M <messagename> "subject line"
    #  message
    #  text
    # .

    # %h = hostname
    # %sys = command from a system() action
    # %act = show list of actions taken preceded by "The following actions were taken:" if any wer
e taken
    # %actnm = show list of actions taken (except email()'s) preceded by "The following actions we
re taken:"
    #         if any were taken

    # %proc = the process name
    # %pid  = pid of process (ie: if found running for R rule)

13    M newsmsg:
14          M proc:
15                MSG WARN: "Warning: %proc on %h not running" """"
                The %proc daemon on %h was not running

                %act

                This is just a Warning.
                Eddie.
                """"

                MSG ALERT: "Alert: %proc on %h not running" """"
                ALERT: The %proc daemon on %h was not running
                This should be looked at immediately.

                %act

                Eddie.
                """"
```

The important parts of this config file follows.

1. This line defines a new ‘Configuration Group’. All definitions and options defined in this group will be local to the group. It is useful to group together configurations for particular common classes of machines in the one group. In this case, we have defined a group called “News” which will define monitoring for imaginary NNTP News servers.
2. Here we define a notification object called “NEWSALERT”. Notification objects contain levels of notification with each level defining how the end user should be notified during the escalation process which Eddie follows. The levels are usually defined to increase from low importance to high importance so that Eddie can ‘escalate’ the problem if the user has not acknowledged or fixed the problem within a certain time frame.
3. “notifyperiod” defines the notification period for this particular notification object. The period may differ from the global checking period defined in the main config file because you may want to be notified less frequently than the checks are actually performed.
4. “escalperiod” defines the length of time before Eddie escalates the problem to the next level of notification. Eddie will do this if the user has not acknowledged that they have received the message and the problem has not yet been fixed.
5. The “Level” directive defines each level of escalation. As Eddie escalates the problem it will use increasing levels of notification to indicate how to send the messages and what messages to send.
6. This is a method of alert. These can be of various types including ‘email’, ‘page’, ‘sms’, ‘elvin’, etc. There are a few alert methods built into Eddie but more can be easily added by the user.
7. The indentation of messaging methods is important in defining how and when you want the different methods of notification to take place.

   A messaging method indented the same as the method above it is called along with that method. Thus in our ‘News’ example, the methods: “email(alert,WARN),elvin(“News warning...”)” (both methods are called together) and “sms(sys-sms,WARN_P)” are called at the same time when a Level 1 notification is required.

   A messaging method that is indented from the method above is only called if the method above it fails (either the messageing system reports back that the method failed, or an acknowledgement timeout occurs and it is not yet time to escalate to the next escalation level). So in the example the method “page(systems,WARN_P)” would be called if the “sms(sys-sms,WARN_P)” method fails to send the message.

   In the case where two methods are on the same line, both methods have to fail before the alternative method is called. There can be as many alternative messaging methods as the user desires.
8. The checking directives are now defined, beginning with the “PROC” directive. This defines a check that a process is either running or is not running (you can check for either). You follow “PROC” with the process name that you want checked. Following the colon are entries which depend on the current directive. In this case, “PROC” expects the rule (“R” to check for a running process; or “NR” to check that the process is not running) followed by a notification call. The notification call specifies the NEWSALERT object and provides arguments of the message object to send and (optionally) the escalation level to start at. In this “PROC” example, the message object is “newsmsg.fs” and we have specified escalation level 0 as the first level. We will introduce message objects below.

9. The “FS” directive defines a file system check. On this line we specify a check of the filesystem “/opt/local/news” and the rule to check is “capac>=90%”. If this rule fails, the notification object is ‘called’ with the given parameters.
10. “PID” defines a pid-file check. There are two checks which can be performed with pid-files: “EX” checks for the existence of the given pid-file; and “PR” checks that the PID found in the given pid-file is actually a currently running process. If either fail, the notification object is called.
11. “SP” is a service port check. This allows you to define a TCP or UDP port and the address that the port should be listening on, and Eddie will check that this port is alive and listening on the current host.

    In this directive we define a ‘depends’ clause which states that this check depends on the “innd” check of the “PROC” directive. This means that if PROC.innd has failed (i.e.: “innd” was not running) then this check would be redundant and a notification for this check would not be sent.
12. The “COM” directive allows you to define any shell command sequence which will be executed by Eddie and the STDOUT, STDERR and return value captured and provided for rule comparison. In this case, the command “/opt/news/bin/newschk” would be executed, and the check would fail if the return value was not equal to 0.
13. Finally, we need to define message groups and objects. Message groups provide an easy way to group together common messages, and message objects define the text that is actually sent to users via the notification system.

    Here we define a message group called “newsmsg” which simply contains a number of sub-groups. Message groups can contain a mixture of message groups and message objects.
14. This is another message group called “proc” who’s parent object is “newsmsg”.
15. “MSG” defines a message object which contains the actual message text to send to the user. This message object is called “WARN” and contains two strings. The first string would be the subject (for notification methods which use subjects) and the second string is the message body.

    You will notice that the strings contain some words beginning with a percentage (‘%’) symbol. These are variables which are substituted with live information just before the message is passed to the notification method. In this case, “%proc” would be substituted for the name of the process which had failed the check, “%h” would be substituted for the hostname of the machine the check had failed on, and “%act” would be substituted for a list of actions taken by Eddie when it found the check had failed.

Appendix B - Porting Eddie

This appendix will discuss the directory layout of the Eddie distribution and high light which modules need to be created when porting Eddie to a new Operating System or architecture.

The Eddie distribution directory structure looks like [note, some directories may or may not exist in the final distribution]:

```
eddie/
   bin/
      [binary files]
   config/
      [config files]
   doc/
      [doc files]
   lib/
      common/
         [Python modules common to all platforms]
      contrib/
         [Contributed modules and libraries]
      sparc-sun-solaris2.5/
         [Sparc/Solaris 2.5 specific modules]
      sparc-sun-solaris2.6/
         [Sparc/Solaris 2.6 specific modules]
      i486-gnu-linux2.0/
         [Intel/Linux 2.0 specific modules]
```

For our discussion on porting we will concentrate on the eddie/lib/ directory. Under eddie/lib/ you will find a common/ and a contrib/ directory which should contain modules which are common to all platforms and contributed modules which are not platform-specific respectively. The other directories are all specific to a certain platform, and in this example, we support Sparc/Solaris 2.5 and 2.6 and Intel/Linux 2.0.

The platform-specific directory name is obtained from a shell script called "systype" which is located in the eddie/bin/ directory. When Eddie is started, this script is executed and the output (which will be the platform name in the format cpu-arch-os) is appended to eddie/lib/ and this directory is used to read the platform-specific modules.

Thus porting Eddie is as simple as creating a new directory under eddie/lib/ (run "systype" on your machine to get the system type which should also be the new directory name) and porting the modules found in the other directories. Currently the modules required are df.py (gets disk usage out of 'df') netstat.py (gets service port information out of 'netstat') and proc.py (gets process information out of 'ps'). We plan to get most of this information straight out of the kernel in a future version.

A more in-depth porting guide will be included in the final Eddie distribution.

Appendix C - Example of Distributed Reporting

Assume that the Eddie agent has found that the innd process has stopped running on our news server. What happens? Firstly, Eddie produces an Elvin notification containing all relevant information to the problem, this information includes:

- **Hostname** - The host the notification originated from
- **Process** - detail of which process has failed, in this case innd
- **Time** - The time the problem was discovered
- **Message** - A tag which tell consumers how to format information into a human readable format.
- **Alert Level** - This indicates the severity of the problem
- **Desired Action** - The allows consumers to pick which notifications are destined for them. For example the SMS consumer would pick up any page actions and send them to the phone of the on call person.
- **Actions Taken** - Any on-host action Eddie may have taken. Eddie supports restart, renice and system as a default set. (see “Eddie Agent Features”

Once the message has been published into the Elvin system, it is then up to consumers to pass the message on further. We currently have an Eddie consumer which takes all notifications generated by Eddie and executes the desired action.

The page( ) action is implemented as a client for Simple Network Paging Protocol [RFC 1861]. For the SNPP server, we use a terrific little product called Quick Page[1] which you can find the Eddie distribution. Quick Page can connect to your paging provider/s using TAP (aka: IXO PET).

[

1. found at ftp.it.mtu.edu:/pub/QuickPage