System Monitoring, Messaging and Notification

Chris Miles <cmiles@connect.com.au>

To describe system monitoring, messaging and notification, as implemented at Connect.com.au pty ltd.

1.0 Introduction

This paper attempts to describe medium to large-scale system monitoring, messaging and notification, as implemented at Connect.com.au pty ltd. The monitoring section (2.0) describes the Eddie monitoring system, which is developed by system administrators at Connect.com.au. A detailed description of Eddie's functionality is provided, along with information on how to configure Eddie and how to add rules and actions.

Section 3.0 covers messaging using the Elvin notification system. Elvin is developed by a research team at DSTC and provides the messaging framework which Eddie and its related applications use to implement a complete and reliable monitoring and notification system.

Section 4.0 describes the common forms of notification used by Connect, including email, SMS (Short Message Service) and Tickertape.

Finally, section 5.0 describes the way that Eddie, Elvin, and the various applications that provide notifications are connected together to make up Connect's monitoring framework. It is hoped that the information provided by this paper can assist other administrators in building a suitable monitoring solution for their systems.

2.0 Monitoring with Eddie

2.1 What is Eddie?

Eddie is a complete system monitoring tool which is under constant development. Eddie is actually an acronym for Essential Distributed Diagnostic and Information Engine. These buzzwords were chosen to describe a tool which would monitor all hosts on a network and aid the system administrator to diagnose problems as well as alerting when things break.

Eddie was designed and developed by Chris Miles (a system administrator at Connect.com.au) and Rod Telford (formally of Connect, now a system administrator at Ekorp pty ltd.) and is written entirely in Python with the aim of being easily extendable and maintainable as well as platform independent.

2.2 What can Eddie do?

In its current form Eddie is a monitoring tool. It also has the basic elements of an information gathering tool, which means it stores the information it gathers from the server for later use by the user. This information storage can be very useful for diagnosing problems, by allowing the user to look up the state of the machine at an earlier point in time. It also provides a useful database for producing reports about the usage of machines over time.

As a monitoring tool, Eddie is very effective. It has a number of directives which cover the most common types of checks that an administrator would want to watch for on a host. New or custom directives can easily be added, making Eddie an easily customizable utility. Common types of alerting mechanisms are also available and, again, adding or customizing the types of alerts is very easy.

The directives available are listed below:

- PROC: provides checks for running processes. The most common use of this directive is to check if a process has stopped running. You can also check if a process is running, if, for example, you did not want it running on the system.

  More complex checks can also be created, which can be simple to complex rules. These rules are actually executed as Python expressions, so there are no limits to what you can do with these. These checks give you access to every available detail about each running process, e.g., owner, priority, memory usage, cpu usage, running time. A simple PROC rule could be:

  ```
  "vsz > 40000"
  ```

  which would return true if the memory footprint of the process at the current instant in time was greater than 40MB.

- FS: filesystem checks allow you to create rules to check filesystem usage. The rules are Python expressions using any of the following variables:
  - size = size of filesystem
  - used = kB used
  - avail = kB free
  - capac = percentage used
  - useddelta = 'used' difference since last check
  - availdelta = 'avail' difference since last check
  - capacdelta = 'capac' difference since last check
  - fs = device
  - mountpt = mount point

  A simple example to check when filesystem is greater than 90% used would be:

  ```
  "capac > 90"
  ```

  A more complex example to check when percentage used >=97% and there is less than 100MB left, or >=90% used and usage increased by 2% since last check:

  ```
  "(capac>=97 and avail<=100MB) or (capac>=90 and capacdelta>=2)"
  ```

- SP: simple checks to make sure local ports are bound to and listening for connections. tcp or udp ports can be checked, and a bind address can be specified if necessary.

  Example, monitor ftp port bound to wildcard address:

  ```
  #  Port      Bind Addr   Action
  SP tcp/ftp:   *            alert()
  ```

- IF: network interface checks. Allows complex rules (evaluated by Python) to be performed on any existing network interfaces. Available variables are:
  - name = name of interface
  - mtu = mtu value
  - net = network
  - address = address of interface
  - ipkts = input packet count
  - ierrs = input error count
  - opkts = output packet count
  - oerrs = output error count
  - collis = collision count
  - queue = queue count

  Example, input errors > 10%:

  ```
  "100.0*ierrs/ipkts > 10.0"
  ```

- NET: network statistics checks. Complex checks on any network statistic can be performed. All values shown by 'netstat -s' are available as variables.

  Examples:

  ```
  "udpInErrors > 0"
  "100.0*tcpRetransBytes/tcpOutDataBytes > 1.0"
  ```

- SYS: system statistics checks. Most common system statistics are available for checking. The available statistics include load average, number of running/sleeping/zombied/stopped processes, percentage of cpu used for idle/user/kernel/iowait/swap, memory/swap used and available.

  Example, load average above 1:

  ```
  "loadavg1 > 1.0"
  ```

  Example, amount of free swap < 40MB:

  ```
  "mem_swapfree < 40 * 1024 * 1024"
  ```

- PORT: check a tcp port on a remote machine. This directive can be used to simply check if the port is accepting connections, or it can send a given string after connection and check a valid response was returned.

  Example, check if port 8080 on host 10.0.0.1 is responding:

  ```
  PORT  '10.0.0.1':  8080  "\012"  ""   alert()
  ```

- COM: a generic check which will execute a given command and allow checks to be made on the command's output (or stderr output) or return code.

  Example, check return code of some command:

  ```
  COM  "/some/command":  "ret != 0"  alert()
  ```

  Example, check ntp stratum:

  ```
  COM  "/opt/local/sbin/xntpdc -c sysinfo | grep stratum | awk '{
  print $2 }'":
              "int(out) > 3"   alert()
  ```

- STORE: this directive does not actually perform any checks. It is used to tell Eddie to store certain collected system data to a given location. Currently, the only valid location is to 'elvindb' which is a process that Eddie can send data to (over Elvin) so it can store the data in a database.

  The STORE directive is still under development, but most of its functionality is already available.

  Example, store the system statistics data in the 'system' table:

  ```
  STORE: "system.*" elvindb('system')
  ```

Eddie has a number of available actions which are executed if a check is true. Some of these are methods of notification, others are special actions. They are described below:

- email(recipients, msg)

  Send email to the given recipients. msg can be a string or an Eddie MSG object which defines groups of messages with subject and body. Messages can contain references to variables which are substituted before the email is sent. The available variables are dependent on the type of directive.

- system(cmd)

  Executes the command cmd by performing a system() call.

- restart(cmd)

  A restricted version of system(), and only valid for the PROC directive. It allows the current process in question to be restarted by executing the init.d script. It basically calls system() with a command like "/etc/init.d/processname start".

- nice(args)

  Only valid for the PROC directive, allows the nice priority of the process in question to be modified.

- eddielog(args)

  This action simply logs a string in Eddie's logfile.

- ticker(msg)

  Sends an ELVINTICKER message to all Elvin Tickertape processes. An Elvin Tickertape is a program which displays messages on a scrolling strip at the top of the screen. This can alert users faster than email in some cases. msg can be a string or an Eddie MSG object.

- sms(number, msg)

  Send an SMS message to a GSM phone. Eddie will actually send the command to the SMS server via Elvin, and the SMS server will then send the message. msg can be a string or an Eddie MSG object.

- elvindb(tablename)

  Only currently valid for the STORE directive, this action will send the named data to an elvindb process to be stored in a database. The data will be stored in the table named tablename.

2.3 What can't Eddie do (yet)?

There are many planned improvements and additional features for Eddie. These will be added as time permits. Some of these are listed below:

- SNMP: support for SNMP polling and data collecting.

- Escalation: ability to track problem age and escalate a problem if it has not been fixed within a set period of time.
- Fixed problem notification: ability to perform actions when a known problem is fixed.
- Maintenance: a simple method of telling Eddie to ignore certain checks during a period of maintenance.
- History: better history support for collected data and allow this history data to be accessed in rules.
- Dependencies: allow checks to be dependent on other checks, so that a check is ignored if the check it is dependent on has already failed.
- Logfile checks: check logfiles for certain expressions.
- More platforms: port to as many operating systems as possible.

2.4 How to configure Eddie

A configuration file sets out global settings for Eddie as well as defining all the rules that will be checked. The configurations can be spread over multiple files. An 'INCLUDE' command tells Eddie to include other files as part of the configuration.

Global configuration settings are as follows:

- LOGLEVEL=<level>

  the level of detail which will be logged. 0 <= level <= 9
- ADMINLEVEL=<level>

  the level of detail which will be sent to the Eddie Administrator.
- LOGFILE="<filename>"

  define the file to log to.
- ADMIN="<email@address.com>"

  define the Eddie Administrator's email address.
- ADMIN_NOTIFY=<timeperiod>

  set how often the admin log is sent to the Eddie Administrator. timeperiod defaults to seconds, or can be followed by any of the following characters: s (seconds), m (minutes), h (hours), d (days), w (weeks), c (calendar months), y (years).
- INTERPRETERS='<interpreter list>'

  defines a list of interpreters (separated by commas) used by the PROC directive to use the name of the script as the process name rather than the name of the interpreter.
- CLASS classname=<host1>,<host2>,<hostn>

  defines a class of hosts that fall into the same group. These hosts will all be checked by the same check group, as explained below.
- DEF name='<text>'

  define a variable which can be used at any time with the reference $name.
- INCLUDE '<filename>'

  include another file into the config at this point.

The following directives are specified by the keyword, followed sometimes (depending on the directive) by a word or string, followed by a colon. Extra 'argu-

ments' to the directive follow the colon, and can either be on the same line or on a following line. If they are on a following line, they should be indented, as per the Python indentation rules. Eddie actually calls the Python parser to parse the configuration and attempts to follow the Python layout rules as closely as possible.

- group <groupname>:

  this is simply used to group together a set of directives which are common to a class of hosts. The classes are defined by the CLASS command (above) so that any host that falls into a particular class will perform the directives found in the corresponding group.

  The directives below do not have to be within a group at all. If they are not, however, they will be executed by every host.

  Note: currently the whole configuration is parsed and stored on every host, whether the host belongs to the group or not. When Eddie performs the checking it only chooses directives which are in groups that match the classes the current host is defined in.

- M <msggroupname>:

  this directive is used to group together a set of MSGs. This is done because the N (notification) directive can intelligently choose which MSG to use based on the action that called it and the current severity level of the alert.

  To do this effectively, an M group is setup which contains a number of M subgroups, each with the name of the checking directive it is valid for. Each subgroup contains a number of MSG directives which define the different levels or types of messages to be sent.

  Example:

```
M commonmsg:
  M proc:
    MSG WARN: "Warning: %procp on %h not running"
      """The %procp process on %h is not running
         and should probably be checked."""

    MSG ALERT: "Alert: %procp on %h not running"
      """ALERT: The %procp daemon on %h is not running and
          should be checked immediately."""
```

- MSG <msgname>:

  defines a message. This directive expects two strings to follow it. The first is the subject (for emails) or the body (for most other actions) and the second is usually only used by the email action for the message body. See 'M' example above.

- N <name>:

  this directive creates a notification definition. This definition sets up how the notification will occur, how often it will occur, how long before the severity is escalated (when implemented) and the different levels of severity. Each level of severity defines the actual actions that will be performed, along with alternate actions if some fail.

  The N directive can contain the following settings and sub-directives:

  - notifyperiod=<time>

    [not currently implemented]

    how often to notify about the current problem.

  - escalperiod=<time>

    [not currently implemented]

    how long to wait until the problem severity is escalated to the next level.

- Level <number>:

  defines a level of severity and what actions (and alternative actions) to perform at this level.

  Actions are defined as a list, separated by commas. Alternative actions (if the first set of actions all failed) are similarly set out below the main set, but are indented.

  [Note, alternative actions are not yet used by Eddie.]

  Example:

```
N COMMONALERT:
  notifyperiod=10m
  escalperiod=20m

  # Notification
  Level 0:
    email($ALERT,WARN)

  # Warning
  Level 1:
    email($ALERT,WARN),ticker(WARN_P)

  # Alert
  Level 2:
    sms($SYSSUP,ALERT_P),ticker(ALERT_P)
      email($ALERT,ALERT)

  # Serious Alert
  Level 3:
    email($ALERT,ALERT),sms($SYSSUP,ALERT_P),ticker(ALERT_P)
```

The checking directives (described above) should then fill up the rest of the group definition.

2.5 How to Add Directives and Actions

Directives are reasonably easy to add. To let the configuration parser know you are adding a new keyword to the list of directives you must add an entry to the directives dictionary in $EDDIEROOTDIR/lib/common/config.py. ($EDDIEROOTDIR is commonly /opt/eddie) The keywords dictionaries are located at the bottom of the file. Simply add your keyword to the directives dictionary, with a line like:

```
"NEWDIREC": directive.NEWDIREC,
```

the key is the keyword and the value is a pointer to the class that will be instantiated to create the directive object.

Next you must create the class in the directive.py file ($EDDIEROOTDIR/lib/common/directive.py). Probably the easiest way is to copy one of the other classes and modify it to do what you want to do. The class you create should be a subclass of the Directive class, so that it will inherit all the common parsing and checking functions. The initiator function, __init__(), should firstly call the initiator of the parent class, with the line:

```
apply( Directive.__init__, (self, toklist) )
```

Then it can parse the tokens that are passed to it in toklist. Most of the current directives accept two or three tokens, where the first token is the directive keyword

(a string), the last token is a colon, ':', and the second token (if there are three) is the name of the process, interface, filesystem, or whatever is required by the directive.

The class must have a tokenparser() function which will be called by the parser to parse any input found after the colon (unless a new directive is started). This function is passed a token list, toklist, a list of the types of tokens in toklist, toktypes, and the current indentation level, indent. These are used by your function to parse the rest of the input into other information required by your directive, such as rules and, especially, actions.

Finally, your directive needs a docheck() function. This function accepts the Config object which holds the configuration settings. The docheck() function should perform the actual checking that your directive was created for, if any. Not all directives actually perform a check (eg: STORE) but most do.

If the check fails, docheck() should build a self.Action.varDict dictionary containing any variables you would like to make available to MSG objects (these variables will be substituted for any %VARNAME found in the MSG strings), then call self.doAction(Config). This will call the actions that were specified by the directive configuration entry.

New actions can be added very easily. A new function created in the action class in the file $EDDIEROOTDIR/lib/common/action.py will make this action instantly available. Actions are called from the configuration file as function calls of an action object, with arguments passed as-is from the configuration file.

2.6 Where do you get Eddie?

Eddie has not yet been released as code was being cleaned up and standard functionality completed. A pre-release is currently being tested by the developers at Connect and Ekorp, and Eddie will be released in alpha form shortly after that. A message will be sent to the SAGE-AU mailing list when Eddie is available.

3.0 Messaging with Elvin

3.1 What is Elvin?

Elvin is a publish-subscribe messaging system where producer programs publish notifications on the Elvin network and consumer programs subscribe to notifications based on content. Elvin provides a platform-independent, language-independent, scalable, fault tolerant messaging system which is perfectly suited to tasks such as monitoring and alert notification.

Elvin is a research project at the Distributed Systems Technology Centre (DSTC) and under constant development by a team lead by Bill Segall and David Arnold.

3.2 What can Elvin do?

Elvin can accept and deliver 'packets of information' (notifications) between programs which are part of the Elvin network. Programs that publish notifications to the Elvin network are called producers. Programs that listen for notifications are called consumers. A particular program may be both a consumer and a producer.

The Elvin server was designed for high performance and high availability. The Elvin developers claim that a single Elvin server can handle event volumes of over 10000 events per second, and up to 1000 client connections. By adding a second Elvin server the performance would be significantly increased, along with the availability of the Elvin network. Fault tolerance can easily be built in to Elvin clients so that they will immediately fail over to a secondary Elvin server if the primary server fails.

Elvin provides an easily programmable API for many common languages. This allows application developers to easily generate notifications and subscribe to the Elvin network.

Elvin notifications can consist of a limited set of data types. These currently consist of: string, integer and float.

Messages are distributed only to points where they are needed in a multicast fashion. This is friendly to network bandwidth and means higher throughput can be achieved.

Elvin provides a powerful subscription language giving the programmer freedom to create complex expressions for receiving only the notifications that are required.

Multiple Elvin servers can be setup to share the load of notification distribution in an "Elvin Federation". This increases availability, reliability, management and performance.

3.3 What can’t Elvin do (yet)?

Provide unrestricted size and data types for notification contents. This would allow arbitrary notifications to be created by application programmers and would mean that the Elvin system is not restricting the types of notifications that can be sent through it.

Provide a secure connection between clients and the Elvin server. Currently all connections are unencrypted and unauthenticated. A layer of security added to the Elvin protocol would provide much greater reliability of authenticated notifications and the ability to restrict connections to trusted clients.

3.4 How do you build Elvin consumers and producers?

Building Elvin consumers and producers is relatively easy. Examples shown here will be in Python which is the preferred development language of the Elvin and Eddie developers.

Creating a new Elvin producer is as simple as connecting to an Elvin server and sending a notification. A notification consists of a set of key and value pairs. In Python this is easily represented by a dictionary. A notification created in Python would look something like this:

```
n = { 'TICKERTAPE' : 'Chat',
      'TICKERTEXT' : 'Hello World',
            'USER' : 'chris',
         'TIMEOUT' : 10 }
```

This creates a Tickertape notification which defines the Tickertape group as 'Chat', along with a message, user and timeout value. To send this notification you first need to connect to an Elvin server. A snippet of Python code to do this follows:

```
import Elvin
```

```
n = { 'TICKERTAPE' : 'Chat',
      'TICKERTEXT' : 'Hello World',
            'USER' : 'chris',
         'TIMEOUT' : 10 }

e = Elvin.Elvin( hostname='elvinhost', port=5678 )
e.notify(n)
```

and that is all that is required to publish an Elvin notification. The notify() call publishes the notification, defined in n, to the Elvin network. Any consumers with the appropriate subscription expression would instantly receive this notification.

Subscribing to Elvin notifications is almost as simple. A connection to the Elvin server is created, as above. A subscription expression is created, along with a function which will receive any notifications. This subscription (expression and function) is then registered with Elvin and the program can then wait for notifications.

```
import Elvin,time

def notify_cb(self, sub_id, d_not):
    """Elvin notify callback function."""
    print "Elvin gave us:",d_not

e = Elvin.Elvin( hostname='elvinhost', port=5678 )

subscription = 'exists(TICKERTAPE)'
e.subscribe( subscription, notify_cb )
time.sleep( 99999 )
```

The notify_cb() function is the function that is registered with our Elvin subscription. This function will be called when a notification is received that satisfies the subscription expression. The argument sub_id contains the subscription id, and d_not is a dictionary containing the received notification. The subscription expression, 'exists(TICKERTAPE)', will evaluate to true if the notification contains the text "TICKERTAPE". This is the simplest way to receive a specific notification.

3.5 Where do you get Elvin?

All Elvin software is available from the Elvin homepage:

```
http://www.dstc.edu.au/Elvin/
```

The currently available version is 3.12p3 with p4 being released shortly. The Eddie developers are looking forward to Elvin 4, due for release sometime around the end of 1999.

4.0 Notification via email, SMS and tickertape

4.1 What is email?

See RFC 822!

Email is a commonly-used store-and-forward messaging system. It is unreliable and hence not the best notification method for the delivery of alerts. It is, however, useful for general (non-critical) notifications or for sending detailed information about a problem alongside a more intrusive notification method (SMS, paging or Tickertape).

4.2 What is SMS?

SMS is a mobile phone messaging service, it stands for Short Message Service. The SMS standard allows for a maximum 160 character message (with 7-bit character set, or 140 8-bit characters) to be sent over the GSM network. Nearly all GSM digital mobile phones can receive SMS messages, and most system administrators now carry mobile phones. This means that SMS is an attractive paging and notification method for system administrators.

4.3 How do you send SMS?

Most modern GSM mobile phones allow the user to manually send SMS messages to other phones on the GSM network. For computers to send SMS messages they must have a connection to the GSM network. This connection is achieved by connecting a GSM phone to the computer and using the phone's data connection to send SMS messages.

Most modern GSM mobile phones can be connected to a computer by a data connection (or IrDA) and there are also some GSM devices which are designed to be permanently connected to a computer, such as the Siemens M1 and M20 GSM data terminals. Software for mobile phones and other GSM devices is readily available.

4.4 What is a tickertape?

Tickertape is an application that is supplied by the Elvin developers. It provides a scrolling message bar across the user's display and can be setup to receive any or all groups of 'TICKERTAPE' notifications from Elvin. It provides a more noticable messaging system than email, but relies on the user being present at their workstation.

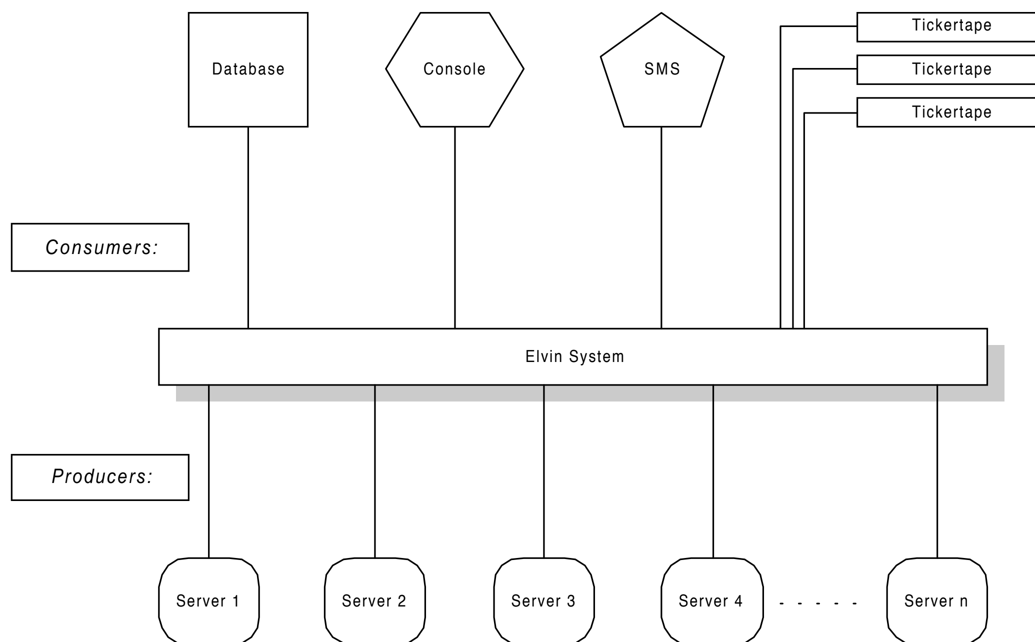
5.0 Connect's monitoring, messaging and notification setup

Connect’s monitoring setup is made up of servers running Eddie to monitor specific elements such as services, processes and disk space. Each of these Eddie processes maintains a connection to the Elvin network where notifications and elvindb data are published.

Various consumers are also connected to the Elvin network. These include an SMS server to accept and send SMS messages, an elvindb database server to accept and store elvindb data, many tickertape programs running on user’s workstations to send and receive messages, and a console to provide information on problems both current and resolved. The console is still in early development but the other programs are all functioning and together make up a complete monitoring, messaging and notification system. A diagramatical view is shown in Figure 1 on page 12.

A detailed description of each of these elements follows.

FIGURE 1. Connect's monitoring, messaging and notification setup



5.1 Servers

Each server runs its own copy of Eddie. There is no central "Eddie Server" that the hosts must rely on, each host performs its own checks and sends out notifications as required.

A centralized Eddie configuration is pushed out to all servers. This means one global configuration is kept in a central location. Any changes are made to this configuration which can then be pushed out to all servers running Eddie. Eddie will automatically re-read the configuration if it notices the files have been modified.

Most servers are placed in groups which share similar rules. This reduces repetition in the configuration files and makes rule management much simpler. Some of the groups are office server, cache, dns server, news server and web server.

Example group: shown below is an example Eddie group for cache servers.

```
group cache:
    ## PROC
    #  Process    Rules Action
    #  -------    ----- ------
    PROC bgp:     NR    COMMONALERT(commonmsg.proc,2)
    PROC named:   NR    COMMONALERT(commonmsg.proc,2)

     # Proxy-related processes
    PROC httpd:   NR   COMMONALERT(commonmsg.proc,2)
    PROC squid:   NR   COMMONALERT(commonmsg.proc,2)
    PROC RunCache:NR   COMMONALERT(commonmsg.proc,2)

    ## FS
    #  Filesystem   Thresholds      Action
    #  ----------   ----------      ----------
    FS /home:       $OVER99         COMMONALERT(commonmsg.fs,0)
```

```
FS /opt:     $OVER95       COMMONALERT(commonmsg.fs,2)
FS /var/log: $VARLOG1      COMMONALERT(commonmsg.fs,2)
FS /var/log: $VARLOG2      COMMONALERT(commonmsg.fs,1)
FS /var/log/cache:$VARLOG1 COMMONALERT(commonmsg.fs,2)

## PID
#    pidfile           Rules Action
#    -------           ----- ------
PID  /var/run/named.pid: EX  COMMONALERT(commonmsg.pid,0)
PID  /var/run/named.pid: PR  COMMONALERT(commonmsg.pidpr,1)

## SP
#   Port        Bind Addr       Action
#   ----        ---------       ------
SP tcp/http:        *           COMMONALERT(commonmsg.sp,1)
SP tcp/http-proxy:  *           COMMONALERT(commonmsg.sp,1)
```

Every server is an elvindb producer. This means that every server sends its collected data to the database (elvindb consumer) via Elvin. See Database (section 5.2) below.

5.2 Database

An elvindb consumer daemon collects all elvindb data sent from the servers (from elvindb producers).

All host data is stored in a database keyed by timestamp and hostname as a record of the system's state at that instant in time.

Data is available for every server in 10 minute slices. The data that is stored by this process is configurable.

Currently available data is:

- system, eg: load average, free memory, swap used, number of running processes.
- netstat, eg: tcpMaxConn, tcpOutDataBytes, tcpInErrs, udpInDatagrams, udpInErrors, ipInReceives, icmpOutMsgs, icmpOutErrors.
- proc: details of every process on the system at a particular point in time, eg: procname, pid, user, priority, memory used.
- interface, details of every network interface at a particular point in time, eg: interface name, mtu, address, in packets, in errors, number of collisions.

A database of host data is a valuable aid in diagnosing problems by providing a means of examining the state of the system before or during the time the problem was occurring. The information stored by this process would not normally be available for the system administrator without such a tool in place to collect and store it.

Similarly, a snapshot of the processes running can be a useful aid after a security breach to aid the administrator in analysing how a hacker may have broken into the system.

The information can also be used to produce reports about the usage of a host over time. Reports and graphs detailing trends of such statistics as average load, available memory, idle cpu and available disk can be invaluable when planning for upgrades or system expansion.

5.3 SMS Server:

An SMSMESSAGE consumer listens for any SMS messages posted to Elvin and sends them as SMS messages via the Siemens M1 GSM terminal.

The SMSMESSAGE consumer daemon builds SMS messages out of Elvin notifications that it receives and sends these via SNPP (Simple Network Paging Protocol, rfc 1861) to an SMS server. The SMS server encodes the message into a packed data string and sends this encoded string to the Siemens M1 GSM terminal connected via a serial port. The Siemens M1 accepts a standard 'AT' modem command set. The Siemens M1 will immediately send the SMS message to the GSM network and return a success or failure code as appropriate. Note that success at this point only means the SMS message was successfully delivered to the SMSC (Short Message Service Centre). There is no indication whether the message was successfully delivered to the destination phone. The GSM protocol provides a reporting mechanism where a report will be sent back to the message originator upon successful delivery, but this is yet to be implemented by this software.

The SMSMESSAGE consumer daemon, the SNPP daemon and the Siemens M1 software are all written in Python and were developed by the Eddie developers.

The Siemens M1 data terminal is a small black-box GSM device. It accepts a standard SIM card and comes with a small aerial. A standard serial cable connects the device to a computer and the device is driven be a standard 'AT' modem command set. The M1 device has recently been superseded by the Siemens M20. More information and purchasing details are available at Oztrak, Ballarat, 03 5330 1184, www.oztrak.com.

5.4 Tickertape

A Tickertape is a lightweight application which scrolls messages along a small strip at the top of the screen. This application is a TICKERTAPE consumer and displays any short tickertape message sent over Elvin.

Many Eddie alerts send Tickertape messages which are instantly displayed on every running Tickertape display to instantly catch user's attentions.

All system administrators at Connect have Tickertape running and are instantly alerted to problems as the notifications are scrolled across the screen. This method alerts administrators faster than email, although email is still sent to provide a more thorough description of the problem.

5.5 Console

An EDDIECONSOLE consumer tracks problems and displays the current status of any outstanding problems that any Eddie process knows about.

Users can view recent history of problems, to find out which administrator, if any, has taken ownership of the problem, and whether the problem has been fixed or escalated to a more senior administrator.

The Eddie Console is still in early development.

6.0 Conclusion

Administrators at Connect have built a full-featured monitoring, messaging and notification system using various tools available. The advantages of Python (quick

development, object-oriented, ease of programming) have allowed the developers to create a complete monitoring tool called Eddie. This tool is easily customizable and extendable, has a powerful configuration language, and is mostly platform independent (except for a few small, specific modules).

Tools for notification delivery and information storage were also developed in Python by Connect administrators. These include an SMS server and a database storage tool for Elvin notification objects.

Elvin was chosen as the messaging system to connect these tools together because it had the functionality required, was lightweight, and had a very simple Python API which meant it could be integrated into the current tools with little effort.

The monitoring system in use at Connect works well in its current form. However it is still under constant development with new features being added periodically in an effort to fulfil every monitoring requirement that Connect's administrators have.