An Overview of the

Nimsoft Log Monitoring Solution

Compiled by:
Nimsoft Technical Services

Table of Contents

Introduction

The Nimsoft Log File monitoring probe (hereafter referred to as logmon) provides monitoring of ASCII-based files. Typically this will be application log files but the probe is also able to monitor information presented via HTML web pages, command output and messages read from Nimsoft queues. The probe searches the specified targets and will find matches of string patterns or regular expressions as defined in the probe's profiles. Nimsoft alarms are generated when contents in the monitored target match the expressions defined. The probe can also extract metric data from the matched log file entry and store this data in the Nimsoft Quality of Service (QoS) database.

What logmon does

An important source of information for an IT operations staff is the wide variety of log-files on the systems they maintain. Checking these files manually is a very time-consuming job, and it may also be a challenge for all members of the staff to be able to interpret all types of messages in all types of logs. The Nimsoft Log File monitoring probe (logmon) can simplify the job for the systems operations staff by:

- Automatically informing about error situations immediately after they have occurred.
- Filtering out the log-file entries that need manual action. Usually the majority of entries in a log-file are not of interest to the daily operations staff. By setting up watchers and filters in logmon, alarms are generated only for the important log-file messages.
- Specifying a more informative alarm message by modifying the original message text, thus helping the operations staff to locate and fix the problem more easily, without requiring assistance from the system specialist.

Logmon can be configured to monitor ASCII log-files in any format. Experience has shown that very few log-files have the same layout. Some files are line-oriented (single-line files like the UNIX system log-files /usr/adm/messages), while other log-files are record-oriented (multiple-line files, like the ones produced by Oracle). The logmon probe monitors both line-oriented and record-oriented log-files effectively, using a powerful regular expression and/or pattern-matching scheme. The probe checks the log-file for new entries at user-configurable, timed intervals, keeping track of the position within the file between each run. This ensures that only one alarm is sent per log-entry, even if the log-file is truncated or wrapped in the meantime.

A single instance of the logmon probe can be configured to monitor multiple log-files. Within each log-file, logmon can be set up to look for occurrences of many different log-file entries with each log-file entries generating a different alarm message, which may contain both text from the original log-file entry and/or user-defined text.

How logmon works

The Logmon probe runs a set of profiles which essentially contains information about:

- What file to monitor.
- How to monitor it.
- What to look for.
- How to report it.

The probe monitors ASCII based log-files, and keeps information about the files being monitored, according to the selected operating mode. This information is used to determine whether the file has been modified or not. If the file has been modified, a set of formatting and watcher rules determine whether or not an alarm message should be generated.

Profile

A logmon profile contains all the necessary information needed to maintain full surveillance of an ASCII log-file. The profile is tagged with a name that describes the purpose of the profile. A log-file may be monitored by one or more profiles each operating independently of the others.

The Watcher definition

A watcher definition instructs logmon as to which message patterns to look for, by using regular expressions or pattern matching rules. You may define multiple watcher definitions to extract various messages from the log-file. If you wish to extract only part of the message (such as a process name, Cluster number etc.), you can define a set of variables that are extracted from the line(s) that matches the watcher rules and the formatting rules, if defined. A variable may be extracted by defining positioning criteria (such as column or character position). There may be cases where you wish to look for messages within a block of lines, and therefore restrict the watchers to the active formatting definition. This is possible within the configuration of a watcher.

The Format definition

The default formatting is to match a single line, however some log-files are record-oriented. To be able to correctly extract information from a record-oriented log-file, logmon needs to determine when a block starts and when it ends. This may be specified using string matching rules and/or a line counter.

For each profile you may choose one of the following monitoring modes:

- Cat
  Used to always do a full read of the logfile from the beginning to the end of the file whether it has been updated or not. This option is often used whilst testing watcher rule definitions.
- Full
  Scans the file from beginning to end only if the file has been modified. Note that the contents of the file must change. The file is not considered as 'modified' unless its contents have changed.
- Full_time
  Similar to the 'full' mode (see above), except that this mode will consider a file to have been

updated even if only it's timestamp has changed. For example, on Unix systems a file that has been "touched" is considered to have been updated.

- Updates
  Scans the file from the previous EOF mark when it's modified. A checksum taken from the end of the file and the file size + file modification time is used to determine whether the file is considered modified or not.
- Command
  Scans the standard console output of a user-specified command.
- URL
  This mode scans the downloaded HTML stream from a specified URL.
- Queue
  Read messages from a Nimsoft message queue instead of from a named file. Messages can be placed on Nimsoft queues via implementations of the Nimsoft API including the Nimsoft SYSLOG Receiver probe (sysloggtw).

Time-formatting primitives

The log-file name/path can be constructed by a combination of text and special primitives (as used by the system-call strftime).

%a -Abbreviated weekday name
%A -Full weekday name
%b -Abbreviated month name
%B -Full month name
%c -Date and time representation appropriate for locale
%d -Day of month as decimal number (01 – 31)
%H -Hour in 24-hour format (00 – 23)
%I -Hour in 12-hour format (01 – 12)
%j -Day of year as decimal number (001 – 366)
%m -Month as decimal number (01 – 12)
%M -Minute as decimal number (00 – 59)
%p -Current locale's A.M./P.M. indicator for 12-hour clock
%S -Second as decimal number (00 – 59)
%U -Week of year as decimal number, with Sunday as first day of week (00 – 53)
%w -Weekday as decimal number (0 – 6; Sunday is 0)
%W -Week of year as decimal number, with Monday as first day of week (00 – 53)
%x -Date representation for current locale
%X -Time representation for current locale
%y -Year without century, as decimal number (00 – 99)
%Y -Year with century, as decimal number
%z, %Z -Time-zone name or abbreviation; no characters if time zone is unknown
%% -Percent sign

For example, to monitor a log-file that switches every day, of the form ddmmyy.log the filename would be specified as:

C:\mylogs\%d%m%y.log

String Matching Methods

Logmon supports two methods of string-matching. The simple method is called pattern matching, which is very similar to the capabilities found in many popular shells (UNIX), and a more powerful method, called regular expressions. The logmon probe implements pattern matching by default, whilst regular expressions are denoted by a / / pair encapsulating the expression.

The capabilities are best highlighted by considering a couple of logmon configuration examples.

Example 1: ASCII Log File

This first example shows an extract from a log file produced by Microsoft Scandisk; this file gets placed in C:\SCANDISC.LOG whenever the user or windows automatically starts scandisk and scandisk detects problems with a disk Cluster.

```
Scandisk could not properly read from or write to Cluster 66293, which contains some or
all of C:\WINDOWS\SYSTEM\MSJET35.DLL.
Resolution: Repair the error
Results: Error was partially corrected.

Scandisk could not properly read from or write to Cluster 121482, which contains some or
all of C:\Program Files\DevStudio\VC\lib\MS
Resolution: Repair the error
Results: Error was partially corrected.
Restarted 10 times because another program wrote to this disk.
```

In order to write a configuration that reacts on the changes in the scandisk.log, and reports this by issuing an alarm, it is required to instruct the logmon probe to search for a block of text that:

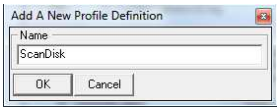
1. Starts with a line containing "`properly read from or write to Cluster`".
2. Ends on a blank line.

The sequence below demonstrates how to define a profile, a formatting definition and a watcher definition using the file extract from SCANDISK.LOG. It reports when it detects data from the file matching the formatting criteria above. If a match is detected data is extracted using the active watcher(s) defined. The purpose is to extract the actual Cluster number and the files involved with the problem, and report it using a user-defined message.

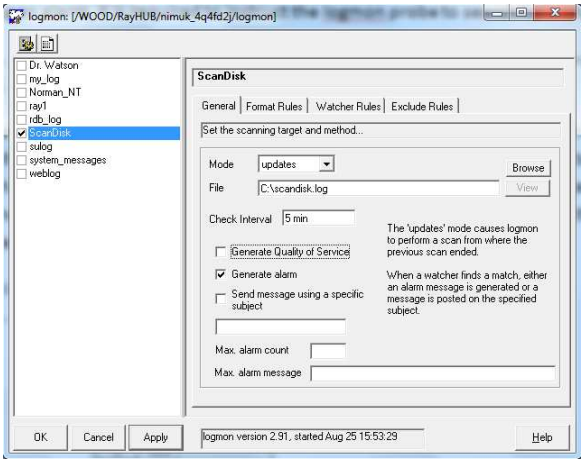
Defining the profile

1. First define which file to monitor, and how to monitor the file.
    a. Click the New Profile button  .

b. Create a profile named ScanDisk and activate the profile.

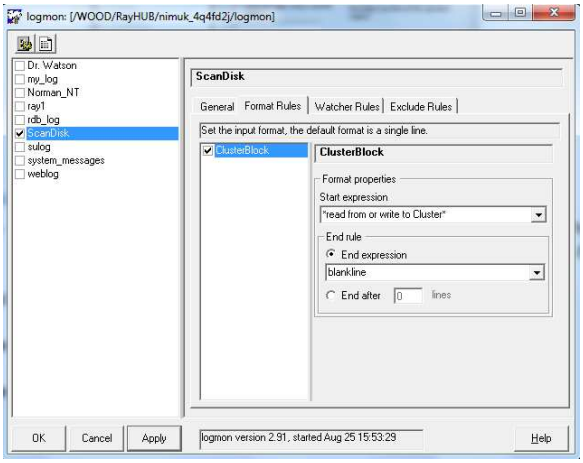


c. Set the File parameter to e.g. C:\SCANDISK.LOG and the Mode parameters to updates.



Defining a formatting definition

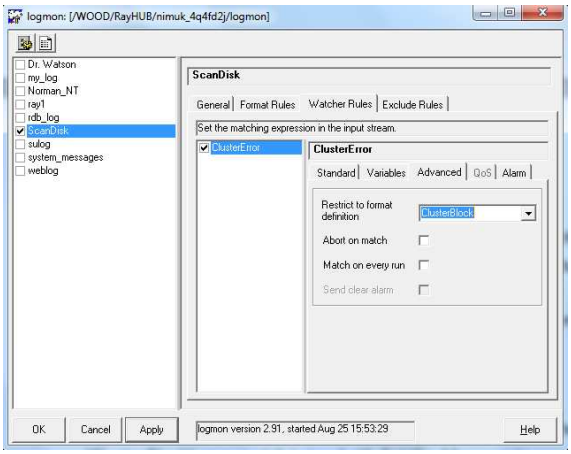
2. Select the Format Rules tab and create a new formatting definition by right-clicking the formats window. Name the definition 'ClusterBlock' and activate the Format Rule.
    a. Replace the Start expr. Field with *read from or write to Cluster*.
    b. Check the End expr. box.
    c. Select blankline from the End expression Combo-box.



Defining the watcher definitions

3. Select the Watcher Rules tab and create a new watcher definition by right-clicking the watchers window. Name the definition 'ClusterError'. Activate the watcher rule.

a. Select Watcher Rules > Advanced tab and select Restrict to watcher rule 'ClusterBlock'.



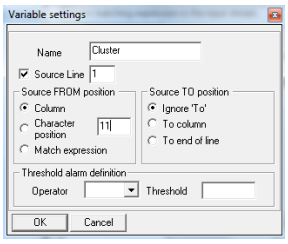
b. Create a new variable definition by right-clicking the variables window in the Watcher Rules > Variables tab.

c. Name this variable 'Cluster'.

d. Check the Source Line and set it to 1 - this tells the watcher to extract the variable from line one within the format stanza.

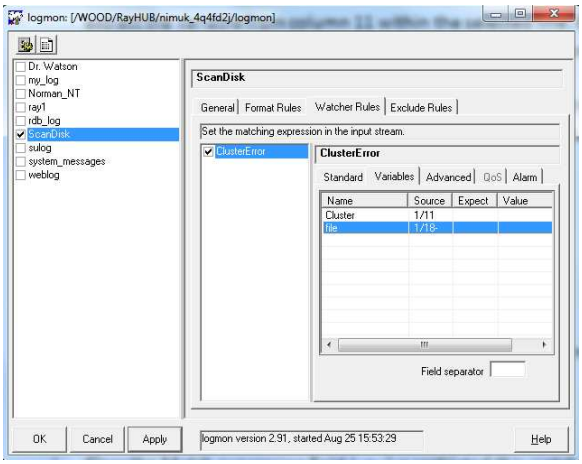
e. Select the column option in the FROM frame, and set it to 11 - this tells the watcher to extract the variable from column 11 within the selected line. By default logmon parses the line using space as a separator character - other separator characters can be specified as required.

f. Select the Ignore 'To' option. This instructs logmon to take whatever's in column 11 on line 1 (one) and store it into a variable that may be referred to as 'Cluster' (or rather $Cluster) in the message string.



g. Create another variable from the data found in line 2, column 7 to the end of the line and call this 'file'.

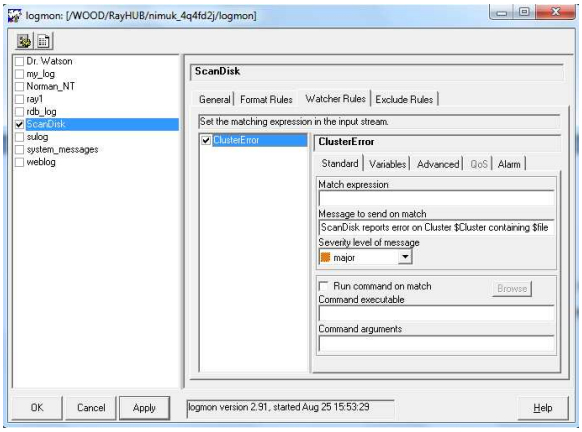
h. Activate both the variables.



i. Select the Watcher Rules > Standard tab.

j. Clear the Match expression field (you've restricted this watcher to the formatting block 'ClusterBlock', so a match is implicit). Set the Message to send on match field to

```
ScanDisk reports error on Cluster $Cluster containing $file.
```

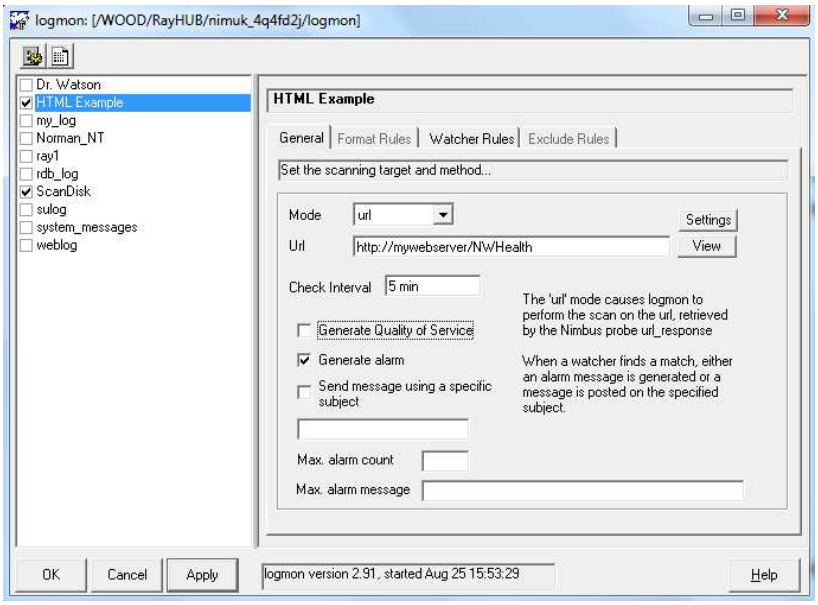


k. Save the definition by clicking the Apply button.

Example 2: HTML page

Regular Expressions are very useful if you need to do more complex analysis of a logfile's contents. In this example we will use regular expressions to parse an HTML page and extract various values.

As in the previous example we first define a profile, but this time we select the "url" mode and specify a URL (http://mywebserver/NWHealth) instead of a file name.



This is a sample of some HTML returned from the example web page:

<TR bgcolor=#efeee9><TD><INPUT TYPE="CHECKBOX" NAME=S_0_0_ CHECKED></TD>

<TD><INPUT TYPE="CHECKBOX" NAME=N_0_0_></TD>

<TD><A HREF="javascript:newWindow('/NWHealth/HELP/WTDRESP', 'Generic','width=600,height=500,resizable=no,screenX=20,screenY=0,scrollbars')"><IMG ALIGN=absbottom BORDER=0 SRC=/SYS/LOGIN/info2.gif

ALT="Item Specific Help"></TD>

<TD><A HREF="DebugInfo">Work To Do Response Time</A></TD>

<TD align=center><IMG ALIGN=absmiddle BORDER=0 ALT="GOOD" SRC="/SYS/LOGIN/good.gif"></TD>

<TD><FONT COLOR=Green>0</FONT></TD>

<TD><FONT COLOR=Green>0</FONT></TD>

<TD><FONT COLOR=Green>0</FONT></TD>

Logmon works with regular expression variables (i.e. the matched values between parentheses within the regular expression) and is able to assign these to logmon variables defined within the watcher rule. We will use this capability to capture these fields:

a. Description - from the "DebugInfo" section
b. Status - from the "IMG" section
c. Current - from the first "COLOR" section
d. Peak - from the second "COLOR" section
e. Max - from the third "COLOR" section

When logmon receives the HTML page all carriage returns and line feeds are removed so the HTML code appears as one contiguous line. Regular expressions are very useful here for breaking the line down into its constituent parts. This is the full regexp used to match the data in the sample:

/<TR[^<]*?<TD[^<]*?<INPUT[^<]*?<\/TD[^<]*?<TD[^<]*?<INPUT[^<]*?<\/TD[^<]*?<TD[^<]*?<A[^<]*?<IMG[^<]*?<\/TD[^<]*?<TD[^<]*?<A[^>]*?>([^<]*?)<\/A[^<]*?<\/TD[^<]*?<TD[^<]*?<IMG[^>]*?ALT=\"([^\"]*?)\"[^<]*?<\/TD[^<]*?<TD[^<]*?<FONT[^>]*?COLOR=(\w+)[^>]*?>([^<]*?)<\/FONT[^<]*?(?:<\/TD[^<]*?)*<TD[^<]*?<FONT[^>]*?COLOR=(\w+)[^>]*?>([^<]*?)<\/FONT[^<]*?(?:<\/TD[^<]*?)*<TD[^<]*?<FONT[^>]*?COLOR=(\w+)[^>]*?>([^<] *?)<\/FONT[^<]*?(?:<\/TD[^<]*?)*/

To make this easier to understand lets break this regexp into component parts:

- Start table row definition
  <TR[^<]*?
- Include
  <TD[^<]*?<INPUT[^<]*?<\/TD[^<]*?
- Notify
  <TD[^<]*?<INPUT[^<]*?<\/TD[^<]*?
- Info
  <TD[^<]*?<A[^<]*?<IMG[^<]*?<\/TD[^<]*?
- Description ($1)
  <TD[^<]*?<A[^>]*?>([^<]*?)<\/A[^<]*?<\/TD[^<]*?
- Status ($2)
  <TD[^<]*?<IMG[^>]*?ALT=\"([^\"]*?)\"[^<]*?<\/TD[^<]*?

  Note about the next three regexps:
  - Since this page has a bug (it fails to close the <TD> with a </TD> when the status is an error) we use the perl-ism (?:close tag)* so that we match even if the tag isn't there. But with (?:, we don't capture the value (we just use it for grouping), so it doesn't count towards the match numbers!

- Current ($3 $4)
  <TD[^<]*?<FONT[^>]*?COLOR=(\w+)[^>]*?>([^<]*?)<\/FONT[^<]*?(?:<\/TD[^<]*?)*
- Peak ($5 $6)
  <TD[^<]*?<FONT[^>]*?COLOR=(\w+)[^>]*?>([^<]*?)<\/FONT[^<]*?(?:<\/TD[^<]*?)*
- Max ($7 $8)
  <TD[^<]*?<FONT[^>]*?COLOR=(\w+)[^>]*?>([^<]*?)<\/FONT[^<]*?(?:<\/TD[^<]*?)*

Performance

In addition to the great features described above this is all achieved with great performance – speed, low resource utilisation and scalability.

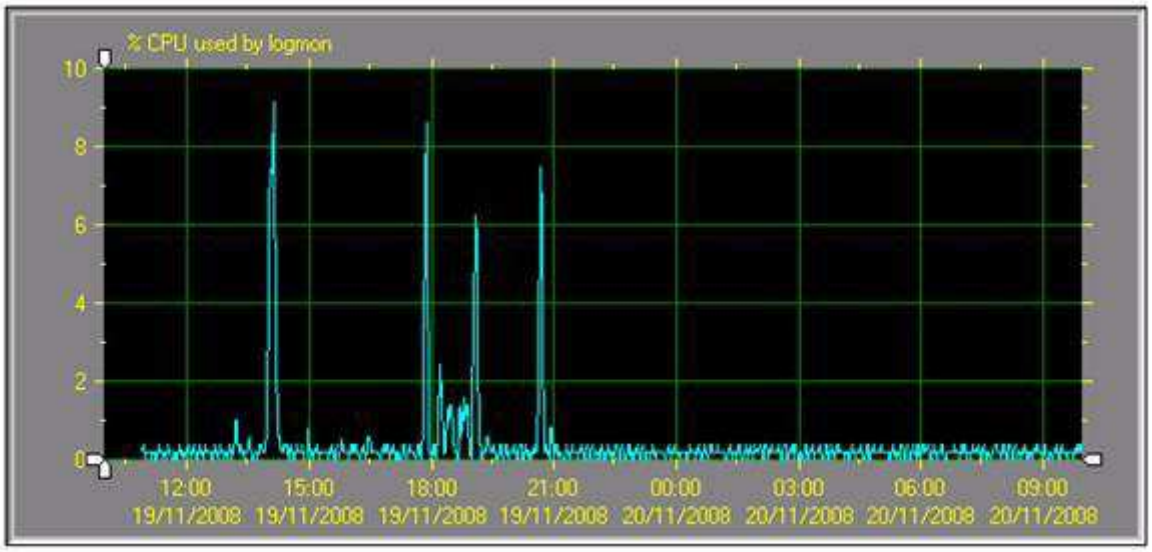
During a recent trial a major investment bank conducted a comparison between their existing log monitoring tool and the Nimsoft Log Monitoring probe.

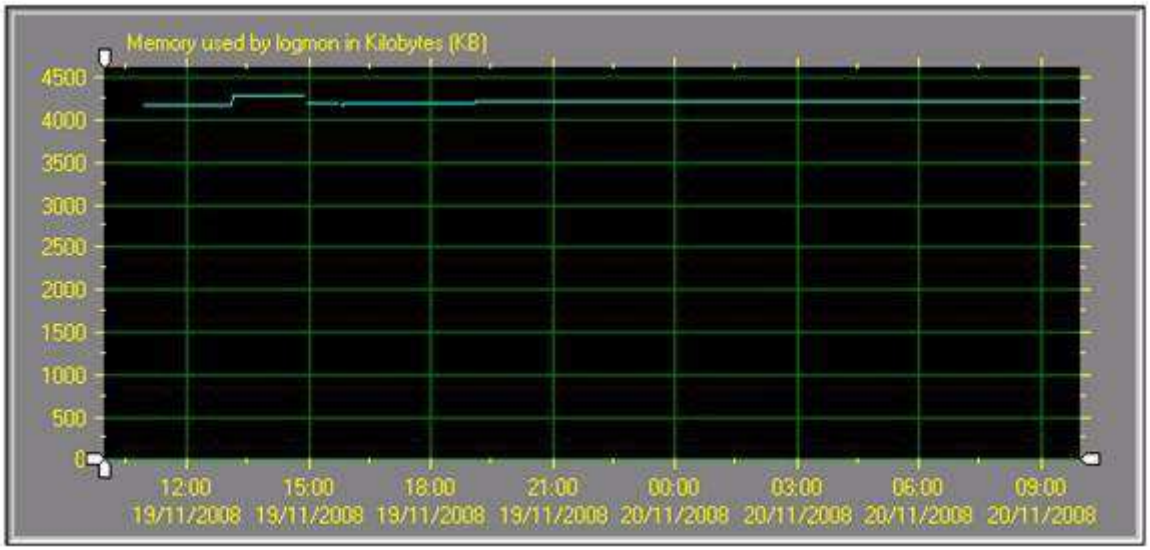
A script was used to convert the existing tools pattern matching definitions into regular expressions within the Nimsoft logmon configuration file format. In total there were approximately 200 regular expression definitions created to monitor messages added to a Unix SYSLOG.

The performance of the logmon process was checked directly against the existing tool. During the test 100 new lines were instantly added to the SYSLOG. The logmon was shown to use considerably less CPU to process these messages:

```
13784 root       5  59    0 4840K 4080K run     0:32 26.37% xxxxx_logfile

13870 root       5  59    0    0K    0K sleep   0:13  3.46% logmon

ROOT@xxxxxxxxx /opt/xxxxxx/lcf/dat/1/xxxxx/bin# ps -ef

    root 13784     1  1 12:05:44 ?        0:33 ./tecad_logfile -c ../etc/xxxxx_logfile.conf

    root 13870 19687  0 12:06:05 ?        0:13 nimbus(logmon)
```

They continued to monitor the performance of the logmon probe during their extensive evaluation process, tracking resource usage over 24 hours (the Nimsoft processes monitoring probe was also used to collect this data). These are the results for CPU and memory:



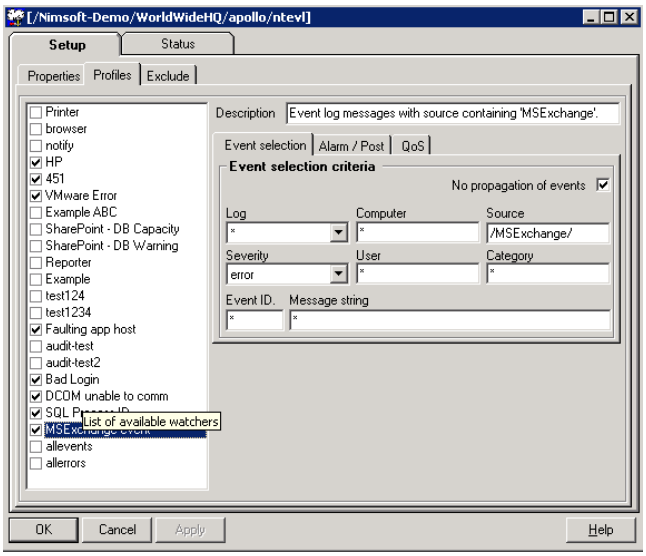


As would be expected CPU usage increased when specific message loads were applied but still well within acceptable limits. Memory utilisation was constant across the complete monitoring period.

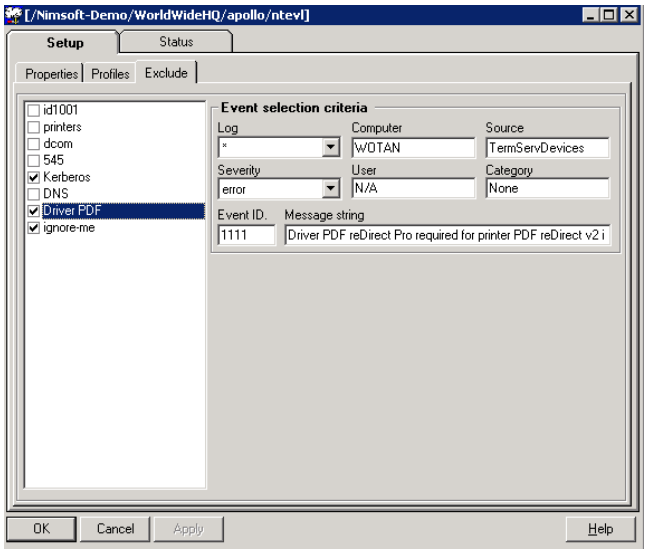
Windows Event Log Monitoring

A companion probe to logmon is the Windows Event Log Monitoring probe (ntevl). Providing similar message filtering capabilities through pattern matching and regular expressions the Windows Event Log Monitoring probe focuses specifically on the Windows Event Log format.

As with the logmon probe, the Windows Event Log Monitoring probe allows for multiple profiles to be defined, each profile looking for a specific set of Event Log messages. The probe understands the specific format of Windows Event Log messages and this is represented in the profile definition. Pattern matching or regular expressions can be used in any field to ensure that only important messages generate a Nimsoft alarm.

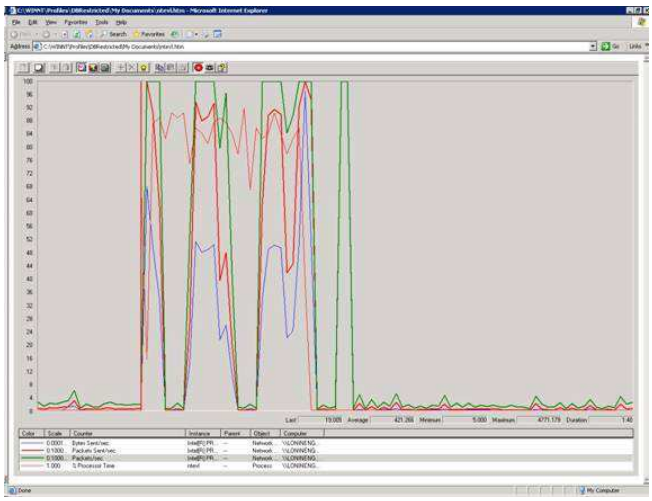


Exclusion profiles allow specific message patterns, which would otherwise be matched through the profiles, to be excluded from generating a Nimsoft alarm.

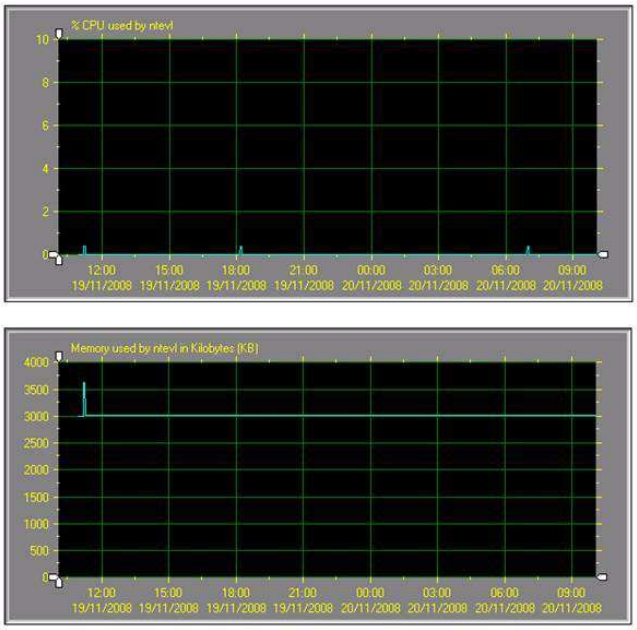


Performance

The same investment bank also evaluated the Windows Event Log Monitoring probe (ntevl). Again they used a script to convert their existing tools log monitoring definitions; over 300 individual profiles (pattern matches) were generated in the ntevl probe configuration. The performance of the ntevl probe was then tested by generating 20,000 Windows Event Log messages within one second. The ntevl monitor was able to process and forward all 20,000 of these events as Nimsoft alarms within 20 seconds of the events being generated. The graph below shows the CPU usage for the ntevl process (thin red line) during the event burst; the other lines show network usage. The ntevl process never consumed 100% CPU.



Once again the Nimsoft processes probe was used to collect resource usage statistics for the ntevl probe over a 24 hour period on a typical system. The following charts show CPU and memory usage for ntevl:



As can be seen, cpu usage of the ntevl probe was minimal, not even reaching 1% over the 24 hour period, whilst memory usage remained constant apart from one small blip.