Linux System Monitor: Usage and Plugin Design Guide

Software User Guide

v.1.0, 12/14/2009, Dan Graham

This document describes how to create additional plugins for the LSM web application. Also, this is a user guide.

Introduction

The basic design of the LSM web application is to allow additional plug-ins to be created on the back-end, so the application can be extended, easily. For example, you can add an additional financial module to monitor currencies flowing through the Linux financial system. Or, you can add additional system modules to monitor some other aspect of system performance, that is not already included in the base installation. This design principle was presented in the detailed design proposal document.

This result of this design is a very versatile, but inflexible, GUI that can present any kind of statistics.

Plugin design involves either writing a program to generate XML from scratch, or inheriting from the probe classes (Perl) that are provided.

The XML is sent over a socket connection to port 9000 on the cvs linux server.

First, the XML syntax will be presented. Then, the steps for inheriting from the probe, contRunningProbe, and sampleProbe classes will be presented.

XML Syntax

The XML syntax is very free-form and is based on the notion of a “topology” describing the elements that are being monitored. This was presented in the detailed design guide. There are four categories of predefined attributes for the XML:

1. An attribute with the same name as the XML node is used to provide a description of the topology element or counter. This description can have spaces. This is the text that will be displayed in the GUI. Without this attribute, it defaults to the name of the XML node.

   Example: <site key="1" **site="Linux site"**>stpaul</site>

2. The “key” attribute specifies a numeric sequence number for a topology element. This is the order the elements will be displayed and aggregated.

Example:

```
<loginItem time="20091208212800" loginItem="loginItem">
    <pty key="3" pty="pty">:0</pty>
    <ip key="2" ip="ip">local</ip>
    <user key="1" user="User Name">dgraham</user>
   <duration counter="avg;max;sum" duration="User Login Duration">  1.1500</duration>
```

</loginItem>

*In this example, the use can see statistics aggregated first by **user**, then by **ip**, then by **pty.***

3. The “time” attribute specifies the measurement time or other associated time with the measurement. A “time” value applies to all descendant XML nodes, and overrides any other “time” specification in a parent node. The time is specified in YYYYMMDDHHMMSS format.

Example:

<loginItem **time="20091208212800"** loginItem="loginItem">
<pty key="3" pty="pty">:0</pty>
<ip key="2" ip="ip">local</ip>
<user key="1" user="User Name">dgraham</user>
<duration counter="avg;max;sum" duration="User Login Duration"> 1.1500</duration>
</loginItem>

4. The “counter” attribute is used to identify an XML node that refers to a numeric value that will be aggregated in the system. There can be more than one “counter” node in the same level of hierarchy. The value of the “counter” attribute is a semi-colon separated list of keywords: avg, sum, min, max. Avg means average values will be calculated for this counter. Sum means the values will be summed. Min means the minimum 10 values will be stored. Max means the maximum 10 values will be stored.

   Implicit in these definitions are that max will always result in a pie chart being displayed with the 10 values. Sum or Avg will result in a time-based bar-chart for lower-granularity times. For example, the daily summary will present a bar-chart showing the AVG or SUM values for each hour in that day. The weekly summary will present a bar-chart showing the AVG or SUM values for each day in the week. The monthly summary will present a bar-chart showing the AVG or SUM values for each week in the month.

Example:

<loginItem time="20091208212800" loginItem="loginItem">
<pty key="3" pty="pty">:0</pty>
<ip key="2" ip="ip">local</ip>
<user key="1" user="User Name">dgraham</user>
<duration **counter="avg;max;sum"** duration="User Login Duration"> 1.1500</duration>
</loginItem>

*In this example, the **avg** keyword will trigger an averaging of login durations for users. So, with the GUI, you can find average login durations for a user, a particular ip address, or a pty. Also, **sum**s are stored so you can see the total login durations aggregated. The **max** keyword triggers pie charts showing the maximum 10 login durations.*

Since, this information was extracted from the “last” UNIX command, it was natural to include

this information of login time and duration.

You don't have to write programs to generate this XML from scratch and send it to the cvs server. Next, the “probe” hierarchy of classes will be presented that allow you to inherit from these classes to create new probes:

PROBE class hierarchy

The following classes are provided: *probe.pm, contRunningProbe.pm, and sampleProbe.pm*

probe.pm

This is the base class for probes, from which the other two classes inherit.

There are two parameters that can be passed to the *new* constuctor:

secs

This specifies the number of seconds between transmissions to the central server. The statistics can be sampled once every *secs,* as in the sampleProbe.pm. Or, in the case of contRunningProbe.pm, the statistics are continually sampled and aggregated and a summary of the statistics are sent to the central server every *secs.*

baseTopo

This parameter specifies a base topology for the particular probe. This is useful for defining a site and server level, for example.

The probe class provides “start” and “stop” member functions that start monitoring and stop monitoring.

The first minimal requirement to build a probe using the “probe.pm” class is to code the “getStats” member function in your derived class. This function is expected to build a statistics hierarchy on the $self->{“statistics”} array or hash reference.

The second minimal requirement to build a probe using the “probe.pm” class is to create the “.conf” configuration file, for specifying which XML nodes are counters, and all of the other pre-defined attributes. This file is a slurped data-structure and the best way to use it is to use an example as a template. See lastProbe.conf or topProbe.conf, for example.

Another class, “statPush.pm” specifies how the XML is pushed to the central server. The current implementation of this is a socket client, but this class could be over-ridden to provide an implementation that writes files to a SFTP directory, for example.

contRunningProbe.pm

You can inherit from this class, if you want to create a probe that scrapes statistics from a terminal-screen oriented continuously running program. This class was used in the topProbe.pm and trafshowProbe.pm implementations.

The following additional constructor parameter is used in this class:

cmd

This specifies the UNIX program to run. This is expected to be a terminal-screen oriented program, such as “top” or “trafshow”.

The first minimal requirement to use this class is to the code the “getSubStats” member function in your derived class. This function is automatically called every time the screen is updated in your running terminal program. The following data structure is used in “getSubStats” to extract information from the screen:

$self->{"vt"}->row_plaintext($i)

Where, $i is the row number in the screen that you wish to query.

The “getSubStats” function can create a measurement topology on the $self->{“subStats”} array or hash reference.

Then, proceed to override the “getStats” function and provide a “.conf” configuration file, like described for the probe.pm class. Except, now the “getStats” function is expected to create a measurement topology in $self->{“statistics”} that is a summary of the work that was done by “getSubStats” in the $self->{“subStats”} structure.

An example is how the “topProbe.pm” class continually aggregates CPU utilization, Memory utilization, swap space, and free memory statistics (about once a second for each time the “top” screen is updated). And, the “getStats” function summarizes (aggregates) this data that is typically sent to the central server once every five minutes.

sampleProbe.pm

You can inherit from this class if you want a probe that runs a UNIX command once every time period and sends statistics based on the output of that command. The following constructor parameters are expected:

cmd

This is the UNIX command that will be run. The text lines of output of this command are stored in the $self->{“cmdResults”} array reference.

The steps for creating a probe from this class are to define the “getStats” function and create the “.conf” file.

The “getStats” function, in this case, is expected to parse through $self->{“cmdResults”} and create the statistics in $self->{“statistics”}

This could be used, for example, to “grep” through Linux output files to generate financial data.

Conclusion

Hopefully, this information will help if any additional probes need to be created. Also, it is hoped that this will further clarify the detailed design of LSM.