Mrut - An Interface between Munin and the PIOSimHD Tool Suite

Alexander Njemz
8njemz@informatik.uni-hamburg.de

Contents

1 Introduction

The objective of this project is to provide a working prototype implementation to trace resource utilization using Munin.

1.1 Munin

Munin [Devb] is a resource monitoring framework. It supports a wide range of plugins which makes it attractive for this project. Munin itself is written in Perl. The plugins are either written in Perl as well or are shell scripts. Munin can be used in a network setting, it is implemented as a client-server architecture. In a typical scenario there are several nodes which are to be monitored. These nodes run the program `munin-node`. `munin-node` is a so called daemon. A daemon is a program that runs as a background process, typically not spawned by the user explicitly but rather by e.g. an init script.

These `munin-node` instances periodically collect information about the machine on which `munin-node` is run. Configuration happens by way of a required `munin-node.conf` file and optionally additional `munin-node` behaviour files, which are put in the `plugin.conf.d` directory.

In addition to the several nodes running `munin-node` there is a monitoring server which runs a program called `munin`.

1.2 Munin Plugins

Munin Plugins are quite numerous. For example, the directory for the munin plugins on our system contains contains 235 plugins. This is the default package installed using the system package manager. For reference, our system is running Fedora Linux 16. Plugins written to be distributed alongside munin are expected to generate useful output when called with the config option. Output values are divided into global attributes and data source attributes.

Some notable global attributes are: `graph_title` and `graph_vlabel`. The `graph_title` attribute is used as the title for the graph generated by the RRDTool. The `graph_vlabel` specifies the label for the y-axis in the generated graph.

Data source attributes pertain to so called fieldnames. For example, the cpu plugin uses fieldnames such as `system`, `user`, `iowait` etc. Interesting data source

attributes are: `label`, `type`, `min` and `max`. `label` supplies the name of the data source. `type` specifies the type (see below) of the data source. The attributes `min` and `max` specify the minimum value, respectively the maximu value, allowed for a data source.

1.2.1 Wildcard Plugins

So called wildcard plugins are just like ordinary plugins but have to be configured before use, by executing `munin-node-configure -shell`, which generates the necessary commands for creating symlinks. The names of wildcard plugins end in `'_'`. For example, there is the `if_` plugin used to monitor the network interfaces available on the system. For each network interface the above command generates a symlink from `if_` to e.g. `if_eth1` etc.

1.3 RRDtool

RRDtool [Devc] is a graphing and logging framework, it is written in C. 'RRD' stands for "Round Robin Database".

We may quote the explanation for the different data source types straight from the rrdtool documentation.

GAUGE

is for things like temperatures or number of people in a room or the value of a RedHat share.

COUNTER

is for continuous incrementing coutners like hte ifInOctets counter in a router.

DERIVE

will store the derivative of the line going forme the last to the current value of the data source. jThis can be useful for gauges, for example, to measure the rate of people entering or leaving a room. Internally, derive works exactly like COUNTER but without overflow checks. So if your counter does not reset at 32 or 64 bit you might ant ot use DERIVE and combine it with a MIN value of 0.

ABSOLUTE

is for counters which get reset upon reading. This is used for fast counters which tend to overflow. So instead of reading them normally you reset them after every read to make sure you have a maximum time available before the next overflow.

1.4 goals

This project aims to provide a library interface to use munin plugins from C code. The code structure as well as basic usage instructions are described in the next chapter.

2 Implementation

In this chapter we will describe firstly the high level concept for the program to be implemented. Next, we will look at the software architecture, i.e. how the functionality sketched in the first part is implemented. And finally, we will describe comprehensively how to use the program.

2.1 Concept

The basic requirements for our tracing software may be stated as follows. It should be possible:

- to provide configuration settings for the tracer in a file
- to provide configuration settings programmatically
- to enable/disable tracing selectively during program runs

The last two points provide the ability to use the tracer not only as a standalone program but also as a library.

A sample program run, eschewing rigor, may be described as follows. First the configuration file is parsed. The configuration file contains entries comprising the absolute path to a plugin and a so called timestep, determinig the time delta between two subsequent collections of the information obtained through the plugin.

The tracer remembers these configuration data.

Next, the tracer runs the specified plugins with the `config` flag and parses the output to set up its internal data structures for storing the information obtained during plugin runs.

Now the tracer may perform its function. Typically, the trace data are output in the format of the HDTraceWritingCLibrary. There are functions for writing output to stdout as well as to files, which is the default mode of operation. In the next section we will look at the implementation details of the tracer.

2.2 Architecture

The main operations performed by the tracer may be put under three headings: parsing, spawning external processes, and threading.

Parsing There are three different parsing tasks. The parsing of the configuration file for the tracer, the parsing of the config output of the individual plugins, and the parsing of the actual plugin output.

For all of these tasks we use the functions from the Lexical Scanner from GLib [Deva] which is a cross-platform utility library maintained by the Gnome foundation.

The configuration file, as mentioned already, consists of entries comprising the absolute path of the plugin to spawn and the timestep, which gives the delay between subsequent runs of the respective plugin. Lines beginning with a # character are treated as comments. The path and the timestep may be separated by any number of tabs or spaces. Here is an example configuration file:

```
/other/mrut/lib/munin/plugins/cpu     1.37
/other/mrut/lib/munin/plugins/memory  2.86

```

The timestep value is supplied in seconds.

The information obtained by parsing the configuration file is kept in the following data structure.

```
struct plugin_info {
    long timestamp;
    long timestep;
    gchar* filename;
    gchar* graph_title;
    gchar* unit;
    GArray *values;
    hdStatsGroup *stats_group;
}
```

The `timestep` field and the `filename` fields keep the aforementioned parsed info. The `timestamp` is used to track the time when the plugin was last executed. The value for the `graph_title` and `unit` fields is obtained executing the plugin with the config option. The `unit` field keeps the value of the attribute `graph_vlabel`. The array `values` is

used to store the actual plugin output values during plugin runs. The `stats_group` field keeps the reference for the value of the stats group for the HDTraceFile.

This is the central data structure for the tracer.

```
struct mrut {
    GArray *infos;
    pthread_t worker;
    pthread_cond_t cond;
    pthread_mutex_t mutex;
    hdTopoNode *topoNode;
    gboolean done;
    GError **error;
};
```

`infos` simply keeps an array of `plugin_info` structs. The `topoNode` member keeps a reference to the `topoNode`.

2.3 Usage

In this section we are going to describe, first of all, the installation, next configuring and running the program and, last, how to visualize the resulting output using HDJumpshot.

2.3.1 Installation

The installation instructions given have been tested on a Linux system but should apply equally to other Unix-like operating systems.

As a dependency the HDTraceWritingCLibrary needs to be installed first. It is assumed that this step has been completed successfully. Instructions of how to install PIOsimHD and its subprojects can be obtained from `http://redmine.wr.informatik.uni-hamburg.de/projects/piosimhd`.

We will suppose that the HDTraceWritingCLibrary is installed in `/opt/lib/HDTraceWritingCLibrary`

The first step to configure the package is:

```
[user@linux: mrut]$ ./waf configure \
--with-hdtrace=/opt/lib/HDTraceWritingCLibrary/
```

Please note that is advisable to choose a sensible prefix. The default is `/usr/local/` which may not be what you want. If no errors occurred the output should be similar to the following.

```
Check for program gcc or cc              : /usr/bin/gcc
Check for program cpp                    : /usr/bin/cpp
Check for program ar                     : /usr/bin/ar
Check for program ranlib                 : /usr/bin/ranlib
Checking for library pthread             : ok
Checking for glib-2.0 >= 2.16            : ok
Checking for library hdTracing           : ok
'configure' finished successfully (0.285s)
```

The next step is to run:

```
[user@hostname mrut]$ ./waf install
```

Note that we assume that the installation directory is writable by the current user. If not, the command is to be executed using `sudo` or as the user `root`

The output should be as follows:

```
Waf: Entering directory ‘/home/user/build/mrut/build’

entering generate conf dir



/opt/mrut
[1/4] cc: src/mrut.c -> build/default/src/mrut_1.o
../src/mrut.c: In function 'helper':
../src/mrut.c:1000:25: warning: initialization from incompatibl
e pointer type [enabled by default]
[3/4] static_link: build/default/src/mrut_1.o -> build/default/
src/libhdMrut.a
[4/4] cc_link: build/default/tools/mrut-tracer_1.o -> build/def
ault/tools/mrut-tracer
* installing include/mrut.h as /opt/mrut/include/mrut.h
* installing munin/lib/plugins/plugin.sh as /opt/mrut/lib/munin
/plugins/plugin.sh
* installing munin/lib/plugins/cpu as /opt/mrut/lib/munin/plugi
ns/cpu
* installing munin/lib/plugins/cpuspeed as /opt/mrut/lib/munin/
```

```
plugins/cpuspeed
* installing munin/lib/plugins/memory as /opt/mrut/lib/munin/pl
ugins/memory
* installing build/default/src/libhdMrut.a as /opt/mrut/lib/lib
hdMrut.a
* installing build/default/tools/mrut-tracer as /opt/mrut/bin/m
rut-tracer
Waf: Leaving directory ‘/home/user/build/mrut/build’
’install’ finished successfully (0.303s)
```

If everything went well, we're now ready to run `mrut-tracer`. First, however, we have to write a config file. The format of the config is as specified above.

The configuration file might look as follows.

```
/other/mrut/lib/munin/plugins/cpu     1.37
/other/mrut/lib/munin/plugins/memory  2.86

```

Before we can run `mrut-tracer` the environment variable `MUNIN_LIBDIR` needs to be set. It has to point to the install directory of the munin plugins, since the plugins rely on a file named `plugin.sh`, which contains utility functions plugins may use. This is accomplished by the following command:

```
[user@hostname ~]$ export MUNIN_LIBDIR=/opt/mrut/lib/munin/plug
ins/
```

If `mrut-tracer` is in our path, we may invoke it simply by:

```
[user@hostname ~]$ mrut-tracer <path-to-configuration-file>
```

if not, we might have to give the full path:

```
[user@hostname ~] /opt/mrut/bin/mrut-tracer <path-to-configurat
ion-file>
```

Note, that if the trace writing library is installed in a non-standard location it might be necessary to modify the environment variable `LD_LIBRARY_PATH`.

Provided that no errors occurred while running `mrut-tracer` the output should look similar to the following:

```
============================================================
file:            /other/build/mrut/lib/munin/plugins/cpu
graph_title:                                      CPU_usage
unit:                                                   %
vals:
 internal:           system        human readable:      system
 min: 0         max: -1         type: TYPE_DERIVE
 internal:             user        human readable:        user
 min: 0         max: -1         type: TYPE_DERIVE
 internal:             nice        human readable:        nice
 min: 0         max: -1         type: TYPE_DERIVE
 internal:             idle        human readable:        idle
 min: 0         max: -1         type: TYPE_DERIVE
 internal:           iowait        human readable:      iowait
 min: 0         max: -1         type: TYPE_DERIVE
 internal:              irq        human readable:         irq
 min: 0         max: -1         type: TYPE_DERIVE
 internal:          softirq        human readable:     softirq
 min: 0         max: -1         type: TYPE_DERIVE
 internal:            steal        human readable:       steal
 min: 0         max: -1         type: TYPE_DERIVE
 internal:            guest        human readable:       guest
 min: 0         max: -1         type: TYPE_DERIVE
------------------------------------------------------------
============================================================
```

After the run of the tracer there are files generated in the directory where `mrut-tracer` is executed with the file extension `.proj`. These can be used for visualization using HDJumpShot.

3 Summary

3.1 Conclusion

The basic objectives as stated have been achieved. However, there are a few fundamental problems, which impede usability.

The first of which shouldn't be too hard to solve in praxis. Currently overflow isn't handled. Recall that munin works in conjunction with RRDTool. In RRD everything is stored internally as type double. The largest representable value for a double (IEEE 64-bit, that is) is roughly $1.7976931348623157 * 10^3 08$. So if an integer counter overflows one can simply add the maximum integer to the stored value. In our solution this is not feasible, since we're storing the values in the appropiate value for HDTrace file format. There's no easy way to simply store everything as a double and supply some sort of conversion function.

The second problem is related to the first one. Munin plugins don't have a notion of data types one could map to C data types in a straight forward way. As mentioned, there is a `graph_vlabel` field. But this is mostly useful for a person interpreting the graphs generated by munin. There's no apparent way to emulate this interpretation in a simple way to C code. Worse, plugin documentation in general is rather poor. To really figure out the meaning of the numbers reported by a plugin one has to read the plugin source code to figure out which kernel interfaces are used and then consult the appropiate documentation.

3.2 Outlook

The first problem mentioned could be solved in various ways. Firstly, by figuring out what the exact use case is for the plugin. E.g. do we want to visualize traces using the HD infrastructure? If yes, it wouldn't be much work to supply the necessary conversion functions to the infrastructure components in question.

The second problem appears generally unsolvable. One solution, if we assume that only one operating system is used and the number of things one wants to trace isn't too big, would be to write the appropiate plugins in C, talking to the kernel directly. This would have the added benefit of obviating the need for any shell or Perl induced overhead.

Bibliography

[Deva] Gnome Developers. Glib. `http://developer.gnome.org/glib/`. 6

[Devb] Munin Developers. Munin. `http://munin-monitoring.org`. 1

[Devc] RRDTool Developers. Rrdtool. `http://oss.oetiker.ch/rrdtool/`. 2