iVoyeur: More Ganglia on the Brain

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is senior systems engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.
dave-usenix@skeptech.org

Welcome to the second installment in my Ganglia series. In the last issue I promised (threatened?) to walk you through building a Ganglia data collection module in C, and that's exactly what we're going to do.

Ganglia, as you'll recall, is composed primarily of two daemons: gmond, which runs on the client side, collecting metrics, and gmetad, which runs server-side and polls the clients, pushing their data into RRDTool databases. Gmond, out of the box, can collect a litany of interesting metrics from the system on which it is installed and can be extended to collect user-defined metrics in three different ways.

The first method to add new metrics to gmond is to collect the data yourself using a shell script, or similar program, piping the resultant data to the Gexec daemon. This method is simple and widely used, but adds overhead to a system carefully designed to minimize it. The second and third methods are to write plug-ins for gmond in Python or C. Writing gmond plug-ins in Python requires that Python be installed and that gmond be compiled against the Python libs on every box where you might want to use your module.

C modules interest me for a number of reasons. Ganglia is such a nifty design that it "feels" wrong to me to burden the system with a bunch of /bin/sh instantiations, to say nothing of the mess implied in the idea of maintaining and scheduling my own shell scripts everywhere. Further, we take great care to minimize the package count in our production environments, and I have no need for Python otherwise. C modules on the other hand are lightweight, elegant, and more likely to make their way into the corporate Git repository so that their versions can be tracked. Finally, a well-written C module can be contributed back to the Ganglia maintainers, and possibly merged into the project, which will benefit everyone who uses it in the future.

How do we write one of these things? In terms of coding strategy, we can organize data collection modules into two types: those I'll call "static" and those I'll call "dynamic." Static modules collect a known series of data metrics. Consider a memory data collector module. We know up front that we're going to be collecting a known series of metrics: free memory, used memory, free and used swap, etc.

Dynamic modules, on the other hand, are going to collect a user-defined series of metrics. The dynamic module I'm going to share with you, for example, is a process-counter module. It simply counts the number of named processes (like httpd)

running on the client system. I'm referring to it as "dynamic" because the module doesn't know which specific processes you want to count beforehand.

Creating a gmond module mostly involves creating a struct of type "mmodule." This struct is made up of three function pointers and another struct containing metric metadata. Because Ganglia relies on the Apache Portable Runtime Library, many of the data structures involved are APR types, and digging more than a couple of layers deep will land you squarely in the APR headers, which you may find a bit vexing (I sure did). The module writers were sensitive to this, and have provided a few functions to insulate you from needing to know too much about the underlying APR data structures. The problem is, to use these functions you need to statically define the metric metadata up front. So the primary difference between writing modules that I call "dynamic" and those I call "static" is how much you'll need to interact with APR. You'll see what I mean as we continue.

Let's take a look at the memory module that comes with the Ganglia distribution, the full source of which is available in the gmond/modules/memory directory of the Ganglia tarball or at http://www.usenix.org/publications/login/2011-02/mod_mem.c. First, take a look at the very bottom of the file:

```
mmodule mem_module =
{
        STD_MMODULE_STUFF,
        mem_metric_init,
        mem_metric_cleanup,
        mem_metric_info,
        mem_metric_handler,
};
```

As you can see, aside from the global STD_MMODULE_STUFF, which is the same for every module, the struct is made up of a pointer to a struct called mem_metric_info, and three pointers to functions, called mem_metric_init, mem_metric_cleanup, and mem_metric_handler. Your module needs to define all three functions, but, as you'll see below, the metric_info struct can be dynamically created later if you're writing a "dynamic" module. This happens to be a static module, so the metric_info struct is statically defined:

```
static Ganglia_25metric mem_metric_info[] =
{       {0, "mem_total", 1200, GANGLIA_VALUE_FLOAT, "KB", "zero", "%.0f",
UDP_HEADER_SIZE+8, "Total amount of memory displayed in KBs"},
        {0, "mem_free", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f", UDP_
HEADER_SIZE+8, "Amount of available memory"},
        {0, "mem_shared", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f",
UDP_HEADER_SIZE+8, "Amount of shared memory"},
        {0, "mem_buffers", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f",
UDP_HEADER_SIZE+8, "Amount of buffered memory"},
        {0, "mem_cached", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f",
UDP_HEADER_SIZE+8, "Amount of cached memory"},
        {0, "swap_free", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f", UDP_
HEADER_SIZE+8, "Amount of available swap memory"},
        {0, "swap_total", 1200, GANGLIA_VALUE_FLOAT, "KB", "zero", "%.0f",
UDP_HEADER_SIZE+8, "Total amount of swap space displayed in KBs"},
#if HPUX
        {0, "mem_arm", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f", UDP_
HEADER_SIZE+8, "mem_arm"},
```

```
        {0, "mem_rm", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f", UDP_
    HEADER_SIZE+8, "mem_rm"},
        {0, "mem_avm", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f", UDP_
    HEADER_SIZE+8, "mem_avm"},
        {0, "mem_vm", 180, GANGLIA_VALUE_FLOAT, "KB", "both", "%.0f", UDP_
    HEADER_SIZE+8, "mem_vm"},
    #endif
        {0, NULL} };
```

Ganglia_25metric is defined in lib/gm_protocol.h. The fields from left to right are:

int key	I'm not sure what this is for, but setting it to zero seems safe.
*char *name*	the name of the metric, for the RRD
int tmax	the maximium time in seconds between metric collection calls
Ganglia_value_types type	used by APR to create dynamic storage for the metric, this can be one of: string, uint, float, double or a Ganglia Global, like the ones above
*char *units*	unit of your metric for the RRD
*char *slope*	one of zero, positive, negative, or both
*char *fmt*	A printf style format string for your metric which MUST correspond to the type field
int msg_size	UDP_HEADER_SIZE+8 is a sane default
*char *desc*	A text description of the metric

Gmond will read the mmodule struct at the bottom of the file and then call the init function contained therein. The init function in turn calls two other important functions: MMETRIC_INIT_METADATA and MMETRIC_ADD_METADATA for each element in the metric_info struct I pasted above. A few APR-related things have been done for us, based on the fact that we defined that metric_info struct, but we don't need to worry about that now, because INIT_METADATA and ADD_METADATA will interact with APR for us. So for each metric listed in our metric_info struct, INIT_METADATA initializes a row of APR storage for our metric and ADD_METADATA calls our handler, which causes data to be collected and stored for our metric.

Examining the handler function, we find that it's a simple switch-case, which is using the index number from the metric_info struct we defined to decide what data to go gather. If you look at any of these data-collection functions (e.g., mem_total_func();), you'll find that they simply read the required information out of the proc file system on Linux.

That's pretty simple. The mmodule struct at the bottom calls init, which calls MMETRIC_INIT_METADATA and MMETRIC_ADD_METADATA once for each element defined in metric_info. MMETRIC_ADD_METADATA in turn calls our handler function, which executes a different function depending on the current index number of the metric_info array. That works for pretty much any physical metric that one can imagine, such as memory, CPU, temperature, or network metrics. All of these metrics are understood up front and may be defined as such.

Things get a little more complicated when we can't predict what the metric_info struct will look like. In the case of my process counter (full source available at http://www.usenix.org/publications/login/2011-02/mod_count_procs.c), an end

user is providing us with the names of one or more processes to be counted, so we won't know until runtime what the metadata in the metric_info struct will look like. Thus, we need to programmatically (or dynamically) generate the metric_info struct at runtime. My mmodule struct definition looks like this:

```
mmodule cp_module =
{
        STD_MMODULE_STUFF,
        cp_metric_init,
        cp_metric_cleanup,
        NULL, /* Dynamically defined in cp_metric_init() */
        cp_metric_handler,
};
```

Since we didn't name the metric_info struct, we need to create it manually in our init function. This requires a bit more interaction with APR. First, I globally declare metric_info at the top of the file:

```
static apr_array_header_t *metric_info = NULL;
```

I also need a Ganglia_25metric pointer inside the scope of the init function, so that I can modify the contents of individual array elements inside metric_info:

```
Ganglia_25metric *gmi;
```

Then I can create the struct using the apr_array_make function:

```
metric_info = apr_array_make(p, 1, sizeof(Ganglia_25metric));
```

p is the name of an APR memory pool that we've been passed from gmond. The second argument is the initial size of the array, and the third argument tells APR what kind of elements this data structure is going to store. It's important that the third element be defined clearly in this manner.

We set gmi to the next available slot in the metric_info stack by calling apr_array _push:

```
gmi = apr_array_push(metric_info);
```

So the general strategy is to read a list of process names from user input and then iterate across them, calling apr_array_push, and manually populating gmi with the metric details we need.

Gmond.conf, the config file for gmond, allows the user to pass parameters to modules with configuration entries such as this one:

```
module {
        name = "cp_module"
        path = "modcprocs.so"
        Param ProcessNames {
          Value = "httpd bash"
        }
}
```

My init function reads these user-supplied parameters using gmond-supplied functions:

```
if (list_params) {
        params = (mmparam*) list_params->elts;
```

```
        for(i=0; i< list_params->nelts; i++) {
          if (!strcasecmp(params[i].name, "ProcessNames")) {
            processNames = params[i].value;
          }
        }
}
```

I use the infamous strtok() function to parse out the space-separated list of process names and iterate through them, first getting a new spot on the metric_info stack and then modifying its contents:

```
for(i=0; processName != NULL; i++) {
        gmi = apr_array_push(metric_info);
        gmi->name = apr_pstrdup (p, processName);
        gmi->tmax = 512;
        gmi->type = GANGLIA_VALUE_UNSIGNED_INT;
        gmi->units = apr_pstrdup(p, "count");
        gmi->slope = apr_pstrdup(p, "both");
        gmi->fmt = apr_pstrdup(p, "%u");
        gmi->msg_size = UDP_HEADER_SIZE+8;
        gmi->desc = apr_pstrdup(p, "process count");
```

Now that the element has been dynamically created, it can have INIT_METADATA, and ADD_METADATA called against it, just like its static brethren:

```
MMETRIC_INIT_METADATA(gmi,p);
MMETRIC_ADD_METADATA(gmi,MGROUP,"cprocs");
```

Once we've iterated across every process the user wants us to count, we put an empty terminator on the metric_info array with one last call to apr_array_push, and then we manually set our metric_info array to be the array used by the mmodule struct down at the bottom of the file (the one we'd initially declared as void) with this line:

```
cp_module.metrics_info = (Ganglia_25metric *)metric_info->elts;
```

Unlike the handler function in the memory module, which calls a different function for each metric in metric_info, my handler always does the same thing. That is to say, it always counts how many of the named process are running. To do the actual counting I've added a function called count_procs, which uses the procps library and is beyond the scope of this article. I do want to point out that my handler function is still using the same metric_index number that the memory module uses, but instead of using it in a switchcase with a function per index number, it's simply using it to de-reference the process name from the metric_info array:

```
Ganglia_25metric *gmi = &(cp_module.metrics_info[metric_index]);
count_procs(gmi->name);
```

That about covers the workings of both types of modules. Once written, I compiled my module on a Linux box from within the Ganglia tarball gmond/modules/cprocs directory:

```
cc mod_count_procs.c -shared -I/usr/include/proc -I/usr/include/apr/
   -I../../../include/ -I../../../lib -lproc-3.2.7 -o modcprocs.so
```

I included /usr/include/proc for the count_procs() function, which uses procps.

In addition to the configuration I've already mentioned in gmond.conf which tells gmond to execute the module and optionally passes parameters to it, gmond also needs to be told to schedule polling for the metrics by adding metric parameters to a collection group:

```
collection_group {
       collect_every = 80
       time_threshold = 950
       metric {
          name = "httpd"
          value_threshold = "80"
          title = "HTTPD Processes"
       }
       metric {
          name = "bash"
          value_threshold = "100"
          title = "BASH Processes"
       }
}
```

Once the module has been compiled and copied into the modules directory (/usr/lib/ganglia/ by default) and the configuration file updated to use it, gmond will report the new metric, and gmetad will create an RRD for it and display it on the front-end. Speaking of front-ends, as I write this, the Ganglia devs are currently going through a user interface rewrite which will update the UI with many of the features I wished for in my last article, including the ability to modify graphs by passing RRDTool options in a URL. The more I use Ganglia, the more I like it, and I heartily recommend checking it out if you haven't yet had the opportunity.

Take it easy.