Introduction to *Stats-Collector*

Javier Palacios

January 2003

Although not directly involved on performance measurement or improvement, performance monitoring is a very important tool that can help to detect problems at early stages or schedule system upgrades. That's quite obvious. But, how can you collect the data? Every Unix system has tools to measure their parameters, but they are for local usage, and not designed for a network, where servers might be even geographically dispersed.

The realm of networked monitoring is probably ruled by SNMP, despite some other useful tools (as statd and friends). Almost every OS or hardware vendor supplies also SNMP agents that can tell you any parameter you want. But that means maintain multiple agents from multiple vendors, and multiple queries to get the same values, because every vendor use a different MIB. This can be avoided using an open-source agent as net-snmp, recently reviewed on SysAdmin, but experience shows that it might be not straightforward depending on the features you want to compile in. In any case, the non-uniformity is not the main objection against SNMP. As well as the alternatives like statd, it means an agent process listening on every server. That means security implications, but also unnecesary complication. Despite the simple on the name, SNMP is a really complex communication protocol. And also very powerful. So much that we usually forget that the M stands for management and not for monitoring, which is our real interest. The point is, why run an SNMP agent periodically polled from a manager node when we just want the output of vmstat command?

It is clear that SNMP does not fit our needs. It might be an acceptable compromise if we deal with both servers and network equipment and want to get them into the same monitoring framework. But, in any other case, is worthwile to search for an alternative approach. A walk around the net gives you a handful of posibilities, ranging from SNMP alike agents to full monitoring and alerting software. Usually designed for server boxes, they fill much better our requirements. But share the unwanted SNMP characteristic of excesive complexity. It's again due to powerful capabilities and extensible design, both desirable in many cases. But the question arises again, as we want to run vmstat and store the results.

Developing a custom system

At this stage we decide to go for a custom design. Our target is really simple, so it should not be so hard. What do we need? A client and server programs, and a format for the messages, the communication protocol. The main goal is simplicity, so we began with Perl and plain text messages. The message core will be the collected values. Unless we transmit all the measured quantities in one message, we need also include which kind of values are we sending. And the message must contain the coordinates of the sample. They could be retrieved from the source address and time, but is much easier if we include them on the message. So, the protocol is rather obvious, and a message looks like

```
nodename|uptime|32:05:21|14/07/2002|0.11:0.17:0.17
```

We use the unix pipe as a field delimiter. The second field is the identifier for the transmited values (we will call it a metric) and should be a token known to the server. Time and date are the next two fields, sorted in the way that Perl likes. The remaining fields are the values and they are format free as will be parsed by a specific function. They might be pipe or space delimited, a mixture, or almost anything. The last issue related to the message, although it actually affect the client and the server, is the chosen transport protocol. We employ UDP for the task. It has a lower impact on systems and network and a margin of lost messages is acceptable. Moreover, if it is good for SNMP should be good for us.

The client and server are both written in Perl, also honouring simplicity. And, of course, fast developement: they are nearly a cut and paste from the examples on perl manual pages, with some extra code for logging, signal handling and a dirty trick to reload the parser module while the server is running. We are not interested on client-server developement but on metric values parsing. The server reads the metric identifier and dispatchs it using a hash of function pointers. There must be one function for each metric, and all of them live on a separate file, which is a perl module that can be reloaded dynamically, allowing us to extend on the fly the number of recognized metrics. The client is even simpler, and just sends the standard input to the address supplied on command line. The real data collection is done by a set of scripts that generate valid message strings. They are probably the only point where the simplicity motto gets slightly relaxed.

The last thing we need is a data storing method. As happens with the transport protocol, it is a preferences issue, and is also not hard to change. Our choose was to employ RRDtool for the task. It is an evolution of MRTG, also by Tobias Oetiker, and it is a fast and reliable data storage, which also includes graphing capabilities. But its most appealing feature, the one that makes it better for us, is the variable temporal resolution. A single file is a container for multiple sampling intervals, allowing to store useful information up to 10 years using only 60 Kb.

An standard deployment will have a central server which works as collector node, were we need RRDtool installed. Once we have the server listening, we jump to the monitored nodes. We need to run periodically the sampler scripts,

and cron jobs are the best way. We have two extreme approaches, individual jobs for every script, or only one job to bind them. We prefer the second approach, and use a small script which sequentially executes all the collectors, although some metrics (as vmstat or iostat) prefer a continuous run instead of small intervals sampling. To reduce the chance of collisions during arrival, we added to the script a quasi-random delay, to spread over time the execution on different nodes.

Extension of the available metrics

A small number of metrics were initially implemented. They were also tightly bounded to some common Unix commands: uptime, vmstat, df, netstat and iostat. Though very basic, they give a very good overview of the general status of the server. Even at this early stage, it helped us detecting a minor problem in our system to check applications availability. Once we completed a successful test period, we named it stats-collector and decided to enlarge the available metrics over the weaker side, with a couple of netstat based metrics for a full coverage of the network activity. Also some knowledge about the running applications is desirable, so we work on the top command output, to construct a summary of a selected set of processes. And the set is completed with a highly specialized metric to query the status of our pure-ftp server, using a command line utility included with the server software. Every metric has an associated sampler script, which shares also the simplicity goal. The complex ones (actually two out of nine) are those who construct summaries and are written in Perl, which makes easier manipulations. But most of them are shell scripts, which typically just write the message header in front of the Unix command output.

The set of measurable metrics is easily enlarged, and the best way to show it is to disect one of them: the netstat_-i metric. Yes, it runs netstat with flags -i, which shows status and statistics for the physical interfaces. This is a shell sampler with very moderate post-processing, and makes it a good and short example. The first lines are shared by every sampler, and define the sample coordinates (the header)

```
DATE=‘date +%S:%M:%H\|%d/%m/%Y‘
HOST=‘hostname‘
HEADER="$HOST|netstat_-i|$DATE"
```

and the remaining part makes the real measurement. In this case the metric name is not actually used on the message but the interface name. The real identifiers are not any problem as far as they are not duplicated.

```
netstat -in -f inet | awk ’
/^(hme|qfe|ge|eri)/ {print $1,$5,$6,$7,$8,$9,$10 }
’ | while read if data ; do
   echo "$HOST|$if|$DATE|$data"
   done
```

The output is sent to udpclient.pl, which opens the socket and push a message like

```
nodename|hme0|29:50:19|24/01/2003|836831728 187482 1238380652 0 0 0
```

This on the client side. But every metric should be recognized and parsed by the server, so we need some coding there. The server splits the message by '|' occurrences and extracts the metric identifier. This identifier is used as index for the dispatch table which points to parser functions. From the server side, this metric is among the most complicated ones, even though it does not mean any complication at all. The message identifier is the name of the interface, so we get a bunch of functions, one for each interface name. That one is the increased complication.

```
$dealer{'hme0'}  = sub {
   nic("hme0",@_);
   };
```

The function which is called is quite standard, and just splits the header and sends the pieces, with the data values to the function that performs the real data storing. We could use also a single function very close to the one described below, just including the interface the interface name in the data string.

```
sub nic {
   my $conf = shift;
   my $host = shift ;
   my $timestamp = &$get_timestamp(splice @_ , 0 , 2);
   my $values = join (":",@_);
   &$write_data($conf,$host,$timestamp,$values);
   }
```

The function write_data is the only place where RRDtool are actually used. It is the only place where many things as disk hierarchy and naming conventions for RRDs appears, and is also the only place to modify if we decide to change the data backend.

RRDs and auxiliar tasks

We have met our initial and simple requirements, relaxing them just a little. But we have said no word about graphs which, by the way, are very importants. And we have also loosely talked about updating RRDs that we have never created. There is a double explanation for that. First of all, RRDtool is an evolving software, with many frontends. Some of them looks simpler, and some of them are able to make much more than our custom system, and it might be a good idea to have a look at them. But the main reason is again the simplicity motto. Built full RRDtool frontend functionality on stats-collector just blows out the goal. But it sounds funny to crow over a simple collector, leaving to others most of the complication and also the major part of real utility. And there is also another important fact. There should be an easy way to set up the system to allow evaluation of the package.

The first task, interfacing with RRDtool is addressed by another in-house developement, which began before stats-collector but has grown in a very close manner. The auxiliar package is called rrdUtils and is in charge to create and graph sets of RRDs. We will describe here only its most important aspects from the stats-collector point of view, submitting to the included documentation for further information. Each metric has an associated configuration file with a list of the nodes where the metric is collected and a description of the datasources. This configuration file includes also the consolidation functions (resolutions and lengths) to store into the RRD, along the graphs of interest for that metric. Two commands allow the creation or graphing of a metric, either for every node or for a selection of them. The second task, the testbed deployment, is accomplished by three scripts located on a subdirectory located at the servers pack and called rrdConf, which also relays on rrdUtils package. This three scripts are in charge to create the whole set of RRDs for every client (install_confs.sh), construct a raw, but usable, navigation tree (web_index.sh) and one more to be a cron job which redraws the graphs periodically (make_graphs.sh).

The future

Now, we can really say that we have finished the task with a high degree of success. It might be simpler, but is hard to find how. And it could be more powerful, of course, but we did never pretend that. So, we can think on future. Probably, the bigger drawback of the whole system is the use of UDP as transport protocol. It forced the introduction of a random delay on the cron job that sequentially launches the sampler scripts, which can only be avoided by a careful distribution over time and nodes. Our measurements show that random delay up to three minutes has no significant lose up to four metrics on twenty nodes, but it is clearly not a a scalable solution. If lost percentages is higher than desired, some technical solution must be adopted. But one of the advantages of the simplistic approach is that it makes easy to add functionality. As far as we keep UDP as protocol, we can use multicast to transport the messages. This allow some kind of redundancy which only needs an included patch and the perl module IO::Socket::Multicast. The other useful task that stats-collector could perform is alarm generation. Wherever we collect data is easy to place a threshold cross detector. It sounds appealing, and it cames also with pockets plenty of complexity. But that's a not yet written story.