Monitoring: Using distribution tools to detect server and network bottlenecks

Hands-On Admin

When a server goes down, it should be your tools, rather than your users, that tell you. The toolset in this article monitors your server and the services it offers, and generates load analysis figures.

BY CHARLY KÜHNAST

Administrators who collect network and server health data, and visualize that data, are likely to diagnose bottlenecks and potential overload scenarios quickly, and without needing to manually search through logfiles. There are any number of useful monitoring tools available for this task, such as Nagios and Big Sister, which Linux Magazine covered in recent issues [1], [2].

But this article is not about using those tools; instead we will be reverting to utilities that most distributions provide as part of their standard armory. We will be adding a few simple, but effective, Bash scripts to help you keep track of what is going on. The real advantage that this toolbox provides in comparison to major network management tools is that you can configure it to reflect your own IT landscape, and individually modify the applications it contains. But there are more benefits:

- You can generate non-standard statistics.
- Besides the results, you also get to see the raw data, and this allows for multiple analysis approaches.
- Shorter learning curve for the admin.
- To a genuine Linux/Unix enthusiast, scripts are far more attractive than ready-to-run binaries.

Of course we can't just ignore the disadvantages:

- The toolkit takes more time to develop to production level.
- Less features.
- If the toolkit proves its use, its developer will be stuck with maintaining it for the rest of his career.

Livecheck – server response?

Let's cover the most important task first: admins first need to check if all their servers are alive, and if the services that should be running on these servers actually are. A simple Bash script called *simple_livecheck.sh* will be a big help in doing so – but a few directories and files are required to support the script. The working directory is called */usr/local/shellscripts/livecheck*. After creating the working directory, create a subdirectory called *etc* below that level, and use your favorite editor to create a file for each server. The filename must adhere to the following convention: *IP_address_name.SLD.TLD*; this happens to be *10.0.0.2_funghi.gondor.com* in our example. Then add the ports that should be open for normal server operations:

```
25
80
110
```

Follow the same pattern for any other servers you need to monitor; the other server in our example is *10.0.0.12_inn.kuehnast.com*, and the only port we need to monitor for this machine is 119.

Nmap – a Useful Tool

The script loops through a list of files that ping the servers to check if they are alive. In this case – and admins will tend to hope that this is the case – the script then uses nmap [3] version 3.00 to check the individual ports. SUSE Linux 9.0 users will need to watch out for an annoying bug: nmap will not run with root privileges.

Listing 1 shows the Bash script (Perl might provide more elegant solutions for some details – and I'm sure that Michael Schilli, the author of our Perl series, will be shaking his head when he reads this article). The server in our example generates the following messages:

```
Server 10.0.0.2 (funghi.➥
kuehnast.com): ping OK
10.0.0.2: Port 25 is up
10.0.0.2: Port 80 is up
10.0.0.2: Port 110 is up
```

I manually halted Apache before repeating the test, just to double-check:

```
Server 10.0.0.2 (funghi.➥
kuehnast.com): ping OK
10.0.0.2: Port 25 is up
10.0.0.2: Port 80 is down
10.0.0.2: Port 110 is up
```

Exactly what I expected. But even newbie admins will appreciate that echoing to the console is a suboptimal form of alerting. It would make more sense to do without the echo and write a syslog entry instead, because in a real-life situation, the script would not be launched manually, but would run as a cronjob. Come to think of it, an email, SMS or pager alert would be even more practical [4].

Multiple Alert Paths

Your choice of approach to alerting will be driven by the role your server plays. One thing to avoid is alerting every 5 minutes by mail or CB radio when the cronjob is triggered. No matter how cool your cellphone ring tone may be, continuous repetitions are a nuisance. A few modifications to the script are required. When called, it must check:

- If a host or port is down, was it down when the script ran previously?
- If everything is okay, was everything okay last time, or has a dead host or port come back to life?

To prepare for the enhanced *alarm_livecheck.sh* script, the next step is to create a subdirectory called *deadhost* below */usr/local/shellscripts/livecheck*.

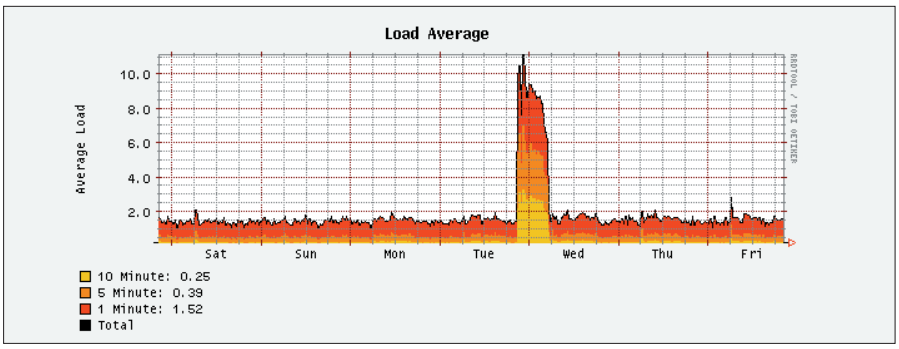

Figure 1: The peak load on the Web server indicates that the server's backup tool wastes system resources. Admins would do well to consider this point when defining alerts. For example, *loadavg > 5.0* for *Alarm* at this point could set your cellphone off every night while your backup job is executing

When the script finds a dead host, it simply creates a file with that host's IP in this directory. In case of a dead service, the script calls the file *IP_Port*.

The mere existence of this file (it does not need any content), tells the script whether the server or service has just died, or was down when the script ran previously, if the server or service has just gone back online, or if the server or service was and still is in a good state of health. A ready-to-run sample script is shown in Listing 2 – of course, you can use cron to launch it. The daily availability information that the script provides for your servers and services, including a history, tells you a lot about the stability of individual components and allows you to identify weak points that the people in charge of servers and networks can address by analyzing the logfiles.

Of course the scripts provide lots of leeway for improvement and enhancements that would benefit both your servers' availability and your personal well-being.

Listing 1: *simple_livecheck.sh*

```
01 #! /bin/bash
02
03 # A script to see if the
   server is responding
04 # and available for use
05
06
   WDIR=/usr/local/shellscripts/l
   m-livecheck
07
08 for i in `ls $WDIR/etc/`; do
09
10   ## extract IP and fqdn from
   file name
11   IP=`echo $i|cut -f1 -d"_"`;
12   NAME=`echo $i|cut -f2 -
   d"_"`;
13
14   ## ping host to see if it's
   up
15   PING=$(/bin/ping -c2 -q -w2
   $IP|grep transmitted|cut -f3 -
   d","|cut -f1 -d","|cut -f 1 -
   d"%")
16   if [ $PING -eq " 0" ]; then
17     ## Host is up
18     echo "Server $IP ($NAME):
   ping OK";
19
20     ## now checking the ports
21     for j in `cat
   $WDIR/etc/$i`; do
22
23       RET=`/usr/bin/nmap -r --
   host_timeout 2500 --
   initial_rtt_timeout 2000 -p $j
   $IP|grep $j/tcp|cut -f1 -
   d"/"`;
24
25       if [ -z $RET ]; then
26         echo "$IP: Port $j is
   down";
27         ## Alarm: Port down ##
28       else
29         echo "$IP: Port $j is
   up";
30       fi
31     done
32
33   else
34     echo "Server $IP ($NAME):
   no response";
35   fi
36 done
```

Server Load & False Positives

The next thing admins are interested in, is how hard their servers are working. There are a few tools that come to mind, such as MRTG [5], RRDTool, and Cacti. I opted for the latter (version 0.6.8), combined with RRDTool 1.0.40. Cacti allows a fairly flexible configuration and is not too complicated; refer to [6] for a detailed description. Cacti provides both network and system load statistics (Load Average), and these are the most important parameters if you are interested in system health statistics. Load graphs allow you to recognize abnormal system behavior, such as peak loads, at a glance. Cacti recently helped yours truly out of spot after I had shot myself in the foot (metaphorically speaking) with a well-meant script. The script in question was a cronjob on my own Web server. It read */proc/loadavg* at five minute intervals, warning me if the average load increased to over 8.0 within a certain period. This happened last Tuesday night. The Cacti graph shown in Figure 1 indicates that the system load peaked abruptly.

The graph that shows the network interface load is well within normal limits, so I immediately ruled out an abrupt increase in HTTP or FTP access. As the

Web server does not run anything worthy of note, apart from those two services, this left only one potential load hog candidate: the backup process. To cut a long story short, I now use a backup solution that is more careful with my system resources.

Collating Network Load

One of Cacti's important features is its ability to collate the total load for one or multiple network interface(s) – and it is a Good Thing to be aware of this load. Wouldn't it be better to know how the load is spread across individual services?

We will again use *funghi.kuehnast.com* as our test server. The server runs Web, mail and POP3 services. Let's assume that Cacti indicates an abnormally high load on the network interface, the first question you would ask is, "Which one of these services is to blame?". Normally, you would need to dig down into the system and find the culprit by analyzing various logfiles. But it would make more sense to have a tool that displayed the network load per port. Again, we could use Cacti for this job, but there is another tool that provides more flexibility.

Multitalented IPTraf Tool

The talented all-rounder, IPTraf [7], is the tool I opted for, as it provides detailed information on network interfaces, including both the current network load per interface, packet sizes, and a traffic overview; also, IPTraf sorts these statistics by port. We use IPTraf version 2.7.0 in our scenario.

Most admins tend to use IPTraf interactively for an overview of the current network traffic on a server. Fortunately, IPTraf can also run as a background daemon. Here, IPTraf logs its results (usually to */var/log/iptraf*) for later parsing. Let's put IPTraf in the cron file first:

```
*/5 * * * * /usr/sbin/iptraf ➥
-s eth0 -t 5 -B -L ➥
/var/log/iptraf
```

The *-s* parameter tells IPTraf to collect traffic information and sort it by port. *-t 5* is the runtime in minutes, following which IPTraf should terminate and write its results to the logfile. *-B* suppresses interactive mode and launches IPTraf as a daemon. IPTraf creates logfile entries similar to the following:

```
TCP/25: 169107 packets, ➥
90804448 bytes total; ➥
96958 packets, ➥
```

Listing 2: *alarm_livecheck.sh*

```
01 #! /bin/bash
02
03 # A script to see if the
   server responds
04 # and on an error, will raise
   the alarm
05 # via Mail/SMS/Cityruf
   Cityruf is similar to 2sms.com
06
07
   WDIR=/usr/local/shellscripts/l
   m-livecheck
08
09 for i in `ls $WDIR/etc/`; do
10
11   ## extract IP and fqdn from
   file name
12   IP=`echo $i|cut -f1 -d"_"`;
13   NAME=`echo $i|cut -f2 -
   d"_"`;
14
15   ## ping host to see if it's
   up
16   PING=$(/bin/ping -c2 -q -w2
   $IP|grep transmitted|cut -f3 -
   d","|cut -f1 -d","|cut -f 1 -
   d"%")
17   if [ $PING -eq " 0" ]; then
18     ## Host is up
19     echo "Server $IP ($NAME):
   ping OK";
20
21     ## check if host was down
   and has now returned
22     if [ -e $WDIR/deadhost/$IP
   ]; then
23       echo "Server $IP ($NAME)
   came back to life";
24       rm $WDIR/deadhost/$IP;
25     fi
26
27     ## now checking the ports
28     for j in `cat
   $WDIR/etc/$i`; do
29
30       RET=`/usr/bin/nmap -r --
   host_timeout 2500 --
   initial_rtt_timeout 2000 -p $j
   $IP|grep $j/tcp|cut -f1 -
   d"/"`;
31
32       if [ -z $RET ]; then
33         echo "$IP: Port $j is
   down";
34
35         ## check if Port was
   down before
36         if  [ -e
   $WDIR/deadports/$IP_$j ]; then
37           echo "Port $j on
   server $IP ($NAME) is still
   dead";
38         else
39           echo "Port $j on
   server $IP ($NAME) has just
   died";
40           touch
   $WDIR/deadports/$IP_$j;
41           ## place commands
   for sending alarm here ##
42         fi
43       else
44         echo "$IP: Port $j is
   up";
45         ## check if port was
   down and has now been
   resurrected ##
46         if  [ -e
   $WDIR/deadports/$IP_$j ]; then
47           echo "Port $j on
   server $IP ($NAME) came back
   to life";
48           rm
   $WDIR/deadports/$IP_$j;
49         fi
50       fi
51     done
52
53   else
54     echo "Server $IP ($NAME):
   no response";
55
56     ## check if Server has
   been dead before
57     if [ -e $WDIR/deadhost/$IP
   ]; then
58       echo "Server $IP ($NAME)
   is still dead.";
59     else
60       echo "Server $IP ($NAME)
   has just died.";
61       touch
   $WDIR/deadhost/$IP;
62       ## place commands for
   sending alarm here ##
63     fi
64
65   fi
66 done
```

```
86978452 bytes incoming; ➥
72149 packets, ➥
3825996 bytes outgoing
TCP/110: 20174 packets, ➥
7575496 bytes total; ➥
8251 packets, ➥
360974 bytes incoming; ➥
11923 packets, ➥
7214522 bytes outgoing
```

The *bytes total* values are particularly interesting, and should be saved for later analysis. It also makes sense to place them in a RRD (Round Robin Database), to provide content for histograms. To support this, create a subdirectory called *data*; a file per service and day will be written to this directory. The service name and date are indicated in the filename. For example, *smtp-history.20031115* contains the SMPT traffic data recorded by IPTraf on November 15 2003.

Stats Database for Cacti

And now to RRD: first create a subdirectory called *rrdtool* in your working directory, */usr/local/shellscripts/iptraf*. The database for our sample server will be stored here; the command to create the database is quite long:

```
01 rrdtool create /usr/local/
shellscripts/iptraf/rrdtool/
mailserver.rrd \
02   DS:smtp:ABSOLUTE:600:U:U \
03   DS:pop3:ABSOLUTE:600:U:U \
04   RRA:AVERAGE:0.5:1:600 \
05   RRA:AVERAGE:0.5:6:700 \
06   RRA:AVERAGE:0.5:24:775 \
07   RRA:AVERAGE:0.5:288:797 \
08   RRA:MAX:0.5:1:600 \
09   RRA:MAX:0.5:6:700 \
10   RRA:MAX:0.5:24:775 \
11   RRA:MAX:0.5:288:797
```

A note for people with prior knowledge of RRDTool; in contrast to the SNMP network traffic data, the data source here is not qualified by *COUNTER*, but by *ABSOLUTE*. This reflects the fact that IPTraf sets the counter to zero every five minutes.

Listing 3: *plot_mailserver.sh*

```
01 #! /bin/bash
02
03 sleep 5 # give IPTraf time to
   write its data into the log
   file
04
05 TRAFLOG=/var/log/iptraf
06
   WDIR=/usr/local/shellscripts/i
   ptraf
07 TODAY=$(/bin/date +%s)
08 UDATE=$(/bin/date +%Y%m%d)
09
10 SMTP=$(grep "TCP/25"
   $TRAFLOG|tail -n1|cut -f2 -
   d","|cut -f2 -d" ")
11 POP=$(grep "TCP/110"
   $TRAFLOG|tail -n1|cut -f2 -
   d","|cut -f2 -d" ")
12
13 echo "smtp: $SMTP"
14 echo "pop3: $POP"
15
16 if [ -z $SMTP ]; then
17   SMTP="0";
18 fi
19
20 if [ -z $POP ]; then
21   POP="0";
22 fi
23
24 # archive results
25
26 echo $SMTP >> $WDIR/data/smtp-
   history.$UDATE
27 echo $POP >> $WDIR/data/pop-
   history.$UDATE
28
29 rrdtool update
   $WDIR/rrdtool/mailserver.rrd
   $TODAY:$SMTP:$POP3
30
31 # draw the graph
32
33 rrdtool graph
   /usr/local/httpd/htdocs/protos
   tats/mailserver.gif \
34 --start -86400 \
35 --vertical-label "bytes per
   second" \
36 -w 600 -h 200 \
37
   DEF:smtp=$WDIR/rrdtool/mailser
   ver.rrd
:smtp:AVERAGE \
38
   DEF:pop3=$WDIR/rrdtool/mailser
   ver.rrd
:pop3:AVERAGE \
39 AREA:smtp#00ff00:"SMTP
   traffic" \
40 LINE1:pop3#0000ff:"POP3
   traffic"
```

You can then follow this pattern to add RRDs for your other servers, remembering to replace the *mailserver.rrd* filename and the *DS* entries. Although you only need this longish command once per server, it is practical to script the command, the advantage being that you can immediately start monitoring any new servers you deploy.

A short script called *plot_mailserver.sh* (see Listing 3) takes care of archiving and plotting graphs; Figure 2 shows the results. Similar scripts are easily written for any other services you need to monitor, such as HTTP, FTP, or NNTP. These scripts and the tools described above provide admins with a solid database that facilitate the troubleshooting of network bottlenecks. In my case, I didn't have to wait long for a real-life opportunity to put the toolset through its paces.

Sudden Death

The scenario: A mail server kept on dying sporadically. The chain of events was always the same: first *alarm_livecheck.sh* alerted me to the fact that the SMTP port was no longer responding, almost immediately followed by the POP3 port; some time later, the server responded only intermittently when pinged, before failing to respond altogether. The network load graph generated by Cacti showed increased activity on the interface about 30 minutes before the server bit the dust, but nothing that Postfix should not have been able to handle.

The Loadavg graph was more impressive: it showed the load ramping to 40 and then dropping to zero. These symptoms are typical of machines that have run out of memory and are swapping into a black hole. In fact, the ancient mail server had a mere 64 Mbytes of RAM and 128 Mbytes of swap space. On the other hand, Postfix is not known as a resource hog. This led me to suspect another program, Spamassassin [8], which I had installed just previously.

To test this hypothesis, I launched top and transferred about a hundred messages to the mail server from another machine. QED. A few moments later, Spamassassin had pushed the mail server above the swap limit. But what should I do about it? I had two approaches in mind:

- Artificially slowing Postfix down, to reduce the mail frequency. This would allow the Spamassassin processes to terminate gracefully and free up the memory they used. It would be easy to configure this in Postfix's *main.cf*, although admittedly quite ugly.
- Install more RAM.

Of course I opted for the latter approach. The problem disappeared with the arrival of 512 Mbytes of RAM and 1 Gbyte of swap space. This example just goes to show that troubleshooting is a lot easier if you monitor your memory resources. So let's do exactly that.

Monitoring Memory Use

Keeping to the tried-and-trusted do-it-yourself approach, I decided to write a short script to read the RAM and swap usage from */proc/meminfo*, write it to an RRD, and plot a graph with these values. The first step was to create the RRD:

```
01 rrdtool create /usr/local/
shellscripts/iptraf/rrdtool/
mailmemory.rrd \
02   DS:ram:GAUGE:600:U:U \
03   DS:swap:GAUGE:600:U:U \
04   RRA:AVERAGE:0.5:1:600 \
05   RRA:AVERAGE:0.5:6:700 \
06   RRA:AVERAGE:0.5:24:775 \
07   RRA:AVERAGE:0.5:288:797 \
08   RRA:MAX:0.5:1:600 \
09   RRA:MAX:0.5:6:700 \
10   RRA:MAX:0.5:24:775 \
11   RRA:MAX:0.5:288:797
```

To be tidy, the database will live next to the network load graphs in the *rrdtool* directory we used previously, although IPTraf will not be supplying the raw data. Note that the data source is defined as a *GAUGE* in this case, as it does not reset to zero after reading, in contrast to the IPTraf data seen previously.

Having created the database, the *meminfo.sh* script shown in Listing 4 can now start collecting data and create a graph.

Looking at the Crystal Ball

Of course the archive function in the IPTraf script serves a useful purpose. I intend to use it at a later stage to generate mid-term network load forecasts. Simply looking at the RRDTool graph shows evidence of linear growth over a period of time in the network load data.

This makes forecasting simple. The first step is to generate a trend-line, a hypothetical straight line that runs through the graph to reflect the individual values on the y-axis as closely as possible. The second step is to calculate the gradient, and simply assume that future measurements will lie in the vicinity of the trend-line. This method is useful for short-term forecasts, assuming there is no noteworthy change to ambient conditions and – most importantly – that your systems are not too close to their physical and technical limits.

The network can also drop – in this case, the trend-line will point downward. However, with the notable exception of a few Dotcoms in their final throes, there is no real evidence of decreasing network or bandwidth use. ■

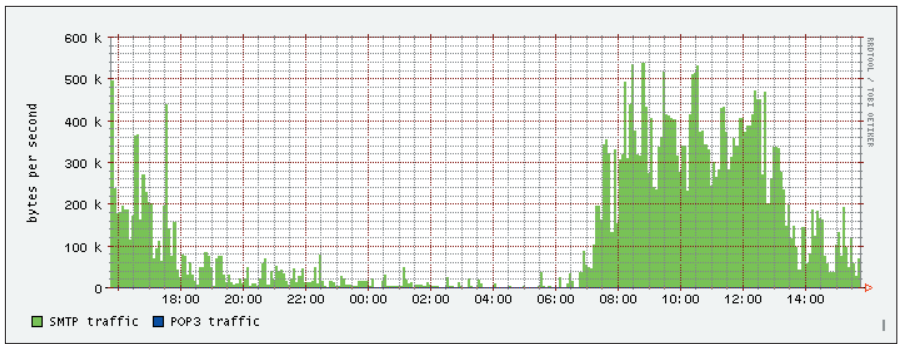

Figure 2: The graph created by the *plot_mailserver.sh* script shows the load on port 25 – and on port 110 actually, although there is no load on this port at present

Listing 4: *meminfo.sh*

```
01 #! /bin/bash
02
03 # A script to test the free
   memory (RAM and swap)
04 # and let RRDTool plot the
   values
05
06
   WDIR=/usr/local/shellscripts/i
   ptraf
07 TODAY=$(/bin/date +%s)
08
09 ## extract mem values from
   /proc/meminfo
10
11 RAM=`grep MemFree
   /proc/meminfo|tr -s
   [:blank:]|cut -f2 -d" "`
12 SWAP=`grep SwapFree
   /proc/meminfo|tr -s
   [:blank:]|cut -f2 -d" "`
13
14 ## write data into the RRD
15
16 rrdtool update
   $WDIR/rrdtool/mailmemory.rrd
   $TODAY:$RAM:$SWAP
17
18 ## draw the graph
19
20 rrdtool graph
   /usr/local/httpd/htdocs/protos
   tats/mailmemory.gif \
21 --start -86400 \
22 --vertical-label "kBytes free"
   \
23 -w 600 -h 200 \
24
   DEF:ram=$WDIR/rrdtool/mailmemo
   ry.rrd:ram:AVERAGE \
25
   DEF:swap=$WDIR/rrdtool/mailmem
   ory.rrd:swap:AVERAGE \
26 AREA:ram#00ff00:"RAM" \
27 LINE1:swap#0000ff:"Swap"
```

INFO

[1] D. Ruzicka, "Network Management with Nagios, Netsaint's Sucessor": Linux Magazine, Issue 29, p. 62

[2] J. Fritsch and T. Aeby, "Always There: Introducing Network Monitoring with Big Sister": Linux Magazine, Issue 38, p. 55

[3] Nmap: *http://www.insecure.org/nmap*

[4] C. Kühnast, "The Sysadmin's Daily Grind: Yaps": Linux Magazine, Issue 30, p. 55

[5] W. Boeddinghaus, "Network Management with MRTG": Linux Magazine Issue 24, p. 56

[6] A. Schrepfer, "Graphical Monitor: Cacti Web Front-end for RRDtool": Linux Magazine Issue 35, p. 55

[7] IPTraf: *http://iptraf.seul.org*

[8] Spamassassin: *http://eu3.spamassassin.org*