Monitoring the CPU loads of all SER processes of B2B agents

Emin Gabrielyan
Switzernet
Updated on 2008-03-26
Created on 2008-02-12

The web site [http://www.unappel.ch/public/080210-ser-cpu/] stores GIF images of graphs showing the SIP servers' CPU loads evolving over time. Each graph shows the CPU loads of all B2B SIP servers on a time axis of about 12 hours. The last, the freshest file, covers a shorter period from the time of its creation until the current time. The last file is extended every 20 minutes until the end of a 12-hour period. New files are created every 12 hours. This document describes the script for creating the graphs and for collecting the data from remote servers.

1. Introduction

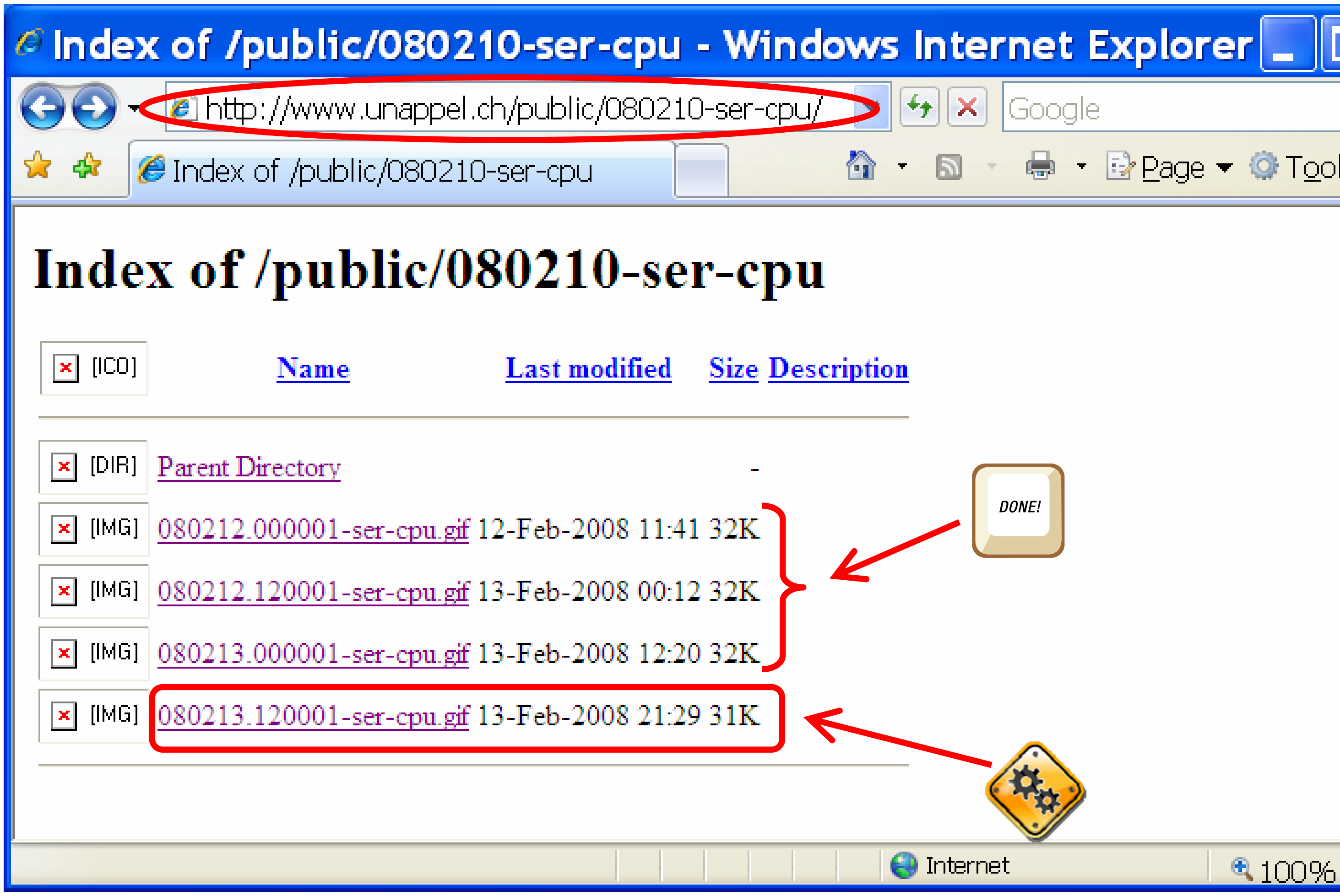
A script running on one of our Geneva computers retrieves the CPU loads of all our SIP servers (serving the customers). The loads are retrieved using the top command executed remotely on each remote machine. Remote execution is made via ssh. For invoking a remote command in a batch mode without an interactive prompt for a password, we use ssh authentication based on public and shared key [SSH login without password].

The CPU load values which are retrieved with a certain periodicity and are stored into a CSV file, are converted into a graph image file. The graph is being updated and is regularly uploaded on the web server while the script is running and is retrieving the CPU loads from the remote machines. The script ends its execution after approximately 12 hours and the last version of the graph on the web server will stay unchanged.

A crontab entry on the Geneva machine launches the script every 12 hours, so a new graph is created twice a day (and is being updated during 12 hours):

```
openser3:~# crontab -l
# m h  dom mon dow   command
...
0 0,12 * * * /root/folders/080212-portasip-ser-cpu/a6.sh.txt > /dev/null
openser3:~#
```

The graph files are uploaded to the web server and are available under the following URL: [http://www.unappel.ch/public/080210-ser-cpu/]. In the URL folder, at any moment, only the last listed graph is being updated (at a periodicity of 20 minutes) while the previous graphs are already completed:


Index of /public/080210-ser-cpu - Windows Internet Explorer
http://www.unappel.ch/public/080210-ser-cpu/
Google
Index of /public/080210-ser-cpu
Page
Tool
DONE!
Internet
100%

Index of /public/080210-ser-cpu

[ICO]	Name	Last modified	Size	Description
[DIR]	Parent Directory		-	
[IMG]	080212.000001-ser-cpu.gif	12-Feb-2008 11:41	32K	
[IMG]	080212.120001-ser-cpu.gif	13-Feb-2008 00:12	32K	
[IMG]	080213.000001-ser-cpu.gif	13-Feb-2008 12:20	32K	
[IMG]	080213.120001-ser-cpu.gif	13-Feb-2008 21:29	31K	

The central program running on the Geneva computer is a bash script [a6.sh.txt]. It is described in details in section 4. With a periodicity of 10 minutes, the script retrieves simultaneously from each SIP server the average CPU load of ser processes. A graph is created, and is being updated and uploaded on the web server with a periodicity of about 20 minutes.

The main bash script uses a Perl script [b7.pl.txt] for converting the CSV files into graphical EPS files (which are further convertible into bitmap files). The Perl script is described in section 3.

2. How to install

You can download and run the program under Cygwin. You need to have installed the lftp tool (command line ftp tool, which makes a part of the Cygwin package) and the Imagemagick program [http://www.imagemagick.org/] for converting the EPS files into GIF or PNG files.

Before running the downloaded bash and Perl scripts, you may need to convert the text files of scripts [a6.sh.txt], [b7.pl.txt] into Unix text format. Use the d2u Cygwin tool for this (use u2d for converting the Unix text back into the DOS format). You must manually create a cpu folder in the current folder (the CSV, EPS, and GIF files are stored in the cpu folder).

For running from your local Cygwin, you may need to edit some of the parameters of the bash script [a6.sh.txt], to uncomment the commented lines, and to store the ftp password in the ftplogin.txt text file:

```
user=sona
domain=youroute.net
hosts="us1,ch1,fr1,fr2,fr3"
topsamp=20
topdelay=30
loop=72
upload=2
delay=20
noerror=1

csv2eps=./b7.pl.txt

convert=/usr/local/bin/convert
localdir=/root/folders/080212-portasip-ser-cpu
passfile=/root/files/070930-unappel-ftplogin.txt

#convert=convert
#localdir=.
#passfile=ftplogin.txt
```

See section 4 for signification of the above shown parameters of the bash script.

2.1. New version

A new version of the bash script is available [a7.sh.txt] since 2008-02-20. It fixes a bug due to which the new machines were not displayed:

```
$ diff a6.sh.txt a7.sh.txt
```

```
6c6
< hosts="us1,ch1,fr1,fr2,fr3"
---
> hosts="us1,ch1,fr1,fr2,fr3,fr4"
89c89
<     foreach $h ("us1","ch1","fr1","fr2","fr3")
---
>     foreach $h (split/,/,"'$hosts'")
```

3. The Perl script converting CSV files into EPS files

The CSV file is a text file containing comma separated values. Below is an example of the content of a CSV file:

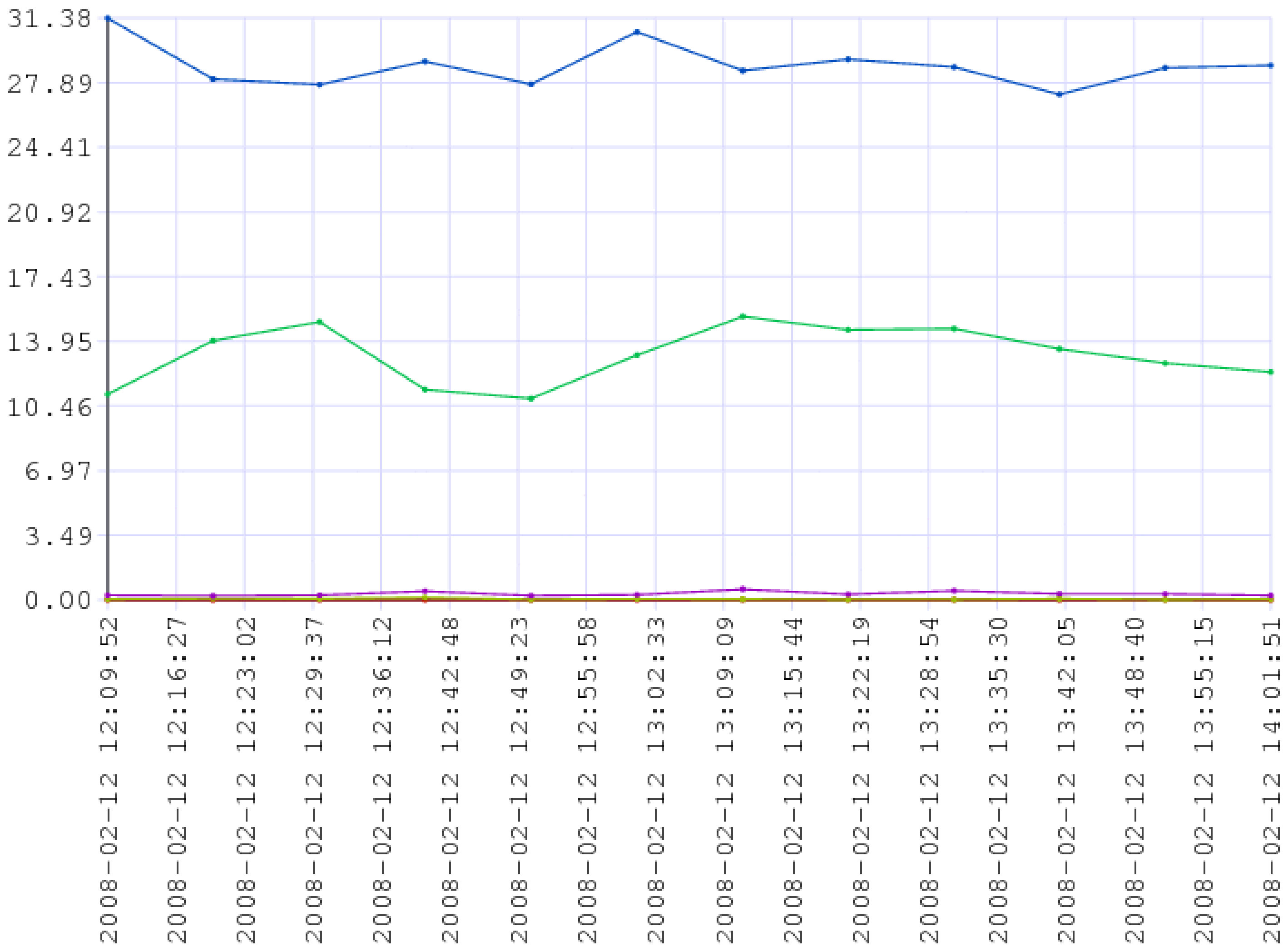
time	us1	ch1	fr1	fr2	fr3
1202814592	0	0.037	11.092	31.3815	0.2415
1202815201	0	0.073	13.9775	28.0975	0.213
1202815817	0	0.059	14.9915	27.7935	0.2475
1202816426	0	0.122	11.3435	29.0465	0.468
1202817039	0	0.022	10.858	27.818	0.222
1202817649	0	0.0515	13.203	30.6365	0.268
1202818260	0	0.0345	15.283	28.551	0.57
1202818869	0	0.0145	14.566	29.1645	0.2945
1202819481	0	0.0145	14.6305	28.741	0.488
1202820090	0	0.061	13.536	27.27	0.32
1202820701	0	0.01	12.7695	28.6995	0.319
1202821311	0	0.0415	12.2925	28.831	0.23
1202821922	0	0.0535	11.459	28.4455	0.5395

[csv]

The CSV file itself can be downloaded and viewed with a text editor (such as Notepad). You may need to convert the file into DOS format (if viewing under Windows) with u2d Cygwin tool.

The EPS abbreviation stands for Encapsulated Post Script. Post Script is a graphical language (often used for printable documents and many printers support the Postscript language). The EPS files can be sent to a postscript printer, can be viewed by MS Word, can be converted into PDF format by Acrobat Distiller, or into PNG, GIF, or JPEG files by the convert tool of Imagemagick [tools].

The example below shows the graph generated by the Perl script [b7.pl.txt] from the above shown CSV [csv] file:



[eps], [gif]

3.1. *Description of the script*

The first column of the below table shows the postscript without modifications and the second column describes the corresponding block of the script:

The Perl Script	**Comments**

`#!/usr/bin/perl`
 	
`$k=1.6;`
 	
`$width=400;`
`$height=200;`
`$xscaleh=140;`
	The parameter $k is the magnification coefficient.
	
	The parameters $width and $height specify the size of the chart area.

```
$yscalew=50;
$fsz=10;
$ptsz=2;
$sample_len=20;
$legendw=75;

$grid_color="0.85 0.85 1";
$xscalen=18;
$yscalen=10;
$yscaledecpt=2;

$width*=$k;
$height*=$k;
$xscaleh*=$k;
$yscalew*=$k;
$fsz*=$k;
$ptsz*=$k;
$sample_len*=$k;
$legendw*=$k;
```

$xscaleh is the height of the area reserved for the labels of the X scale.

$yscalew is the width of the area reserved for the labels of the Y scale.

$fsz is the size of the fonts. $ptsz is the size of the points of curves.

$sample_len is the length of sample curves in the legend area (on the right), and $legendw is the width of the legend area.

$xscalen is the number of vertical gridlines on the X scale (also the number of labels of the X scale).

$yscalen is the number of horizontal lines on the Y scale (also the number of labels of the Y scale).

$yscaledecpt is the number of positions after the decimal point for the values of the Y scale.

```
if(@ARGV != 2)
{
  print "two arguments are required: <csv file name>
<ps file name>\n";
  exit 1;
}

$psfname=$ARGV[1];

$fname=$ARGV[0];

if(! open fh, $fname)
{
  print "error opening file $fname\n";
```

The script requires two arguments.
The first argument must be the input CSV file name and the second argument must be the file name of the output EPS file.

The lines of the input file are read into array @lines.

```
  exit 1;
}

@lines=<fh>;
close fh;
```

```
foreach $_ (@lines)
{
  s/[\r\n]//g;
  @r=split/,/;
```

The loop for checking the format of the input file and for figuring out the range of values being displayed.

The line-feed ‘\n’ and-carriage return ‘\r’ symbols are removed from the lines. The coma separated elements of lines are stored into @r array.

```
  if(/^([\d]+)(,([\d.]*))+$/)
  {
    if(!defined($xmax) || $r[0]>=$xmax)
    {
      $xmax=$r[0];
    }
    else
    {
      print "time is not increasing\n";
      exit 2;
    }
    if(!defined($xmin) || $r[0]<$xmin)
    {
      $xmin=$r[0];
    }
    for($i=1;$i<@r;$i++)
    {
      if(defined($r[$i]) && $r[$i] ne "")
      {
        if(!defined($ymax) || $r[$i]>$ymax)
        {
          $ymax=$r[$i];
        }
        if(!defined($ymin) || $r[$i]<$ymin)
        {
          $ymin=$r[$i];
        }
      }
    }
```

If the line contains data, it must contain the time value followed by at least one comma separated value.

The values are real numbers, or can be empty.

$xmin and $xmax will store the minimal and maximal values of the time scale.

The $ymin and $ymax values will store the minimal and maximal values of the Y scale.

We take care to not consider the empty values as zeroes.

```
  }
```

```
  elsif(/^time(,([\w]+))+$/)
  {
    if(!defined(@h))
    {
      @h=@r;
    }
    else
    {
      print "multiple header lines\n";
      exit 2;
    }
  }
  else
  {
    print "file format error\n";
    exit 2;
  }
```

The title of the first column of the header line must be “time”.

The titles of the following columns can be any non-empty alphanumerical string.

```
}
```

End of the check loop.

```
if(!defined(@h))
{
  print "no header line\n";
  exit 2;
}

if(!defined($xmin) || !defined($xmax))
{
  print "no value on the X scale\n";
  exit 2;
}

if(!defined($ymin) || !defined($ymax))
{
  print "no value on the Y scale\n";
  exit 2;
}

if($xmin==$xmax)
{
  print "only one value on the X scale\n";
  exit 3;
}

if($ymin==$ymax)
{
  print "only one value on the Y scale\n";
  exit 3;
```

After the first scanning loop, the script checks the results.

The script will not attempt to create an output EPS file if any error is encountered.

If the header line is not found the script interrupts.

The script will interrupt also if no data is found, i.e. either $xmin, or $xmax, or $ymin, or $ymax are still undefined.

The graph will not be created also if there is only one value is available along the X or Y axes.

`}`	

`if(! open ps,">".$psfname)`
`{`
`  print "error opening file $psfname\n";`
`  exit 4;`
`}`
`print ps "%!PS-Adobe-3.0 EPSF-3.0\r\n";`
`printf ps "%%%%BoundingBox: 0 0 %f %f\r\n",$yscalew+$width+$legendw,$xscaleh+$height+$fsz;`
	If control checks of the input EPS file are passed successfully, the script opens the EPS file for writing (Prefix “>” before the filename).
	
	The first line contains the EPS header line “%!PS…”.
	
	The second line contains the coordinates of the bounding frame [more on the EPS extension of PS]

`print ps "/Courier findfont $fsz scalefont setfont\r\n";`
`print ps "1 setlinecap\r\n";`
	Specifying the default font settings.
	
	Specifying the shape of the extremities of line (1 is for rounded) [PS bluebook]

`for($i=0;$i<$xscalen;$i++)`
`{`
`  $t=$xmin+$i/($xscalen-1)*($xmax-$xmin);`
`  $x=$yscalew+$i/($xscalen-1)*$width;`
`  print ps "gsave $grid_color setrgbcolor\r\n";`
`  print ps "$x $xscaleh $fsz 3 div sub moveto $x $xscaleh $height add lineto stroke\r\n";`
`  print ps "grestore\r\n";`
`  print ps "gsave\r\n";`
`  print ps "$x $xscaleh translate\r\n";`
`  print ps "90 rotate\r\n";`
 	
`($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime($t);`
`  printf ps "(%s-%02d-%02d %02d:%02d:%02d) dup\r\n",$year+1900,$mon+1,$mday,$hour,$min,$sec;`
`  print ps "stringwidth pop $fsz 2 div add neg $fsz 1 3 div mul neg moveto\r\n";`
`  print ps "show\r\n";`
`  print ps "grestore\r\n";`
`}`
	Drawing the vertical gridlines and the time labels of the X axis.
	
	The time labels format is YYYY-MM-DD HH:MM:SS.

`for($i=0;$i<$yscalen;$i++)`
`{`
`  $v=$ymin+$i/($yscalen-1)*($ymax-$ymin);`
	Drawing the horizontal gridlines and the labels of the Y axis.

```
  $y=$xscaleh+$i/($yscalen-1)*$height;
  print ps "gsave $grid_color setrgbcolor\r\n";
  print ps "$yscalew $fsz 3 div sub $y moveto
$yscalew $width add $y lineto stroke\r\n";
  print ps "grestore\r\n";
  print ps "gsave\r\n";
  print ps "$yscalew $y translate\r\n";
  printf ps "(%.${yscaledecpt}f) dup\r\n",$v;
  print ps "stringwidth pop $fsz 2 div add neg $fsz
1 3 div mul neg moveto\r\n";
  print ps "show\r\n";
  print ps "grestore\r\n";
}
```

```
print ps "$yscalew $xscaleh moveto\r\n";
print ps "$width 0 rlineto stroke\r\n";
print ps "$yscalew $xscaleh moveto\r\n";
print ps "0 $height rlineto stroke\r\n";
```

Drawing the main X and Y axes.

```
sub green
{
  local($a)=($_[0]);
  if($a>=0 && $a<=0.5)
  {
    $a*2;
  }
  elsif($a>=0.5 && $a<=1.5)
  {
    1;
  }
  elsif($a>=1.5 && $a<=2)
  {
    1-($a-1.5)*2;
  }
  elsif($a>=2 && $a<=3)
  {
    0;
  }
  else
  {
    -1;
  }
}

sub red
{
  local($a)=($_[0]);
  $a+=1;
  $a=$a>=3?$a-3:$a;
```

The input CSV file may contain any number of columns. The values of each column must be represented via a distinct curve.

The curves are distinguished by their colors. We rely on a method which can generate a set of distinct colors for any number of curves.

This section specifies a method for generating an RGB color as a function of a scalar value from a range from 0 to 1.

Then, according to the required number of curves, we only need to split the [0,1] range with equally spaced points.

The RGB colors are generated according to the method described in:

```
  &green($a);
}

sub blue
{
  local($a)=($_[0]);
  $a+=2;
  $a=$a>=3?$a-3:$a;
  &green($a);
}

sub rgb
{
  #according to: http://4z.com/People/emin-
gabrielyan/public/050401-linkviews/
  local($a,$k)=($_[0],0.75);
  $a*=3;
  (&red($a)*$k,&green($a)*$k,&blue($a)*$k);
}
```

http://4z.com/People/emin-gabrielyan/public/050401-linkviews/.

```
for($i=1;$i<=@h-1;$i++)
{
  ($rgb_r,$rgb_g,$rgb_b)=&rgb(($i-1)/(@h-1));
  $rgb_color[$i]="$rgb_r $rgb_g $rgb_b";
}
```

Creating an array containing the colors of currently needed curves as provided in the CSV file (using the above defined &rgb function).

```
$y=$xscaleh+$height;
for($i=1;$i<=@h-1;$i++)
{
  print ps "gsave $rgb_color[$i] setrgbcolor\r\n";
  print ps "$yscalew $width add 1.0 $sample_len mul
add $y moveto $sample_len 0 rlineto stroke\r\n";
  print ps "$yscalew $width add 1.5 $sample_len mul
add $y $ptsz 2 div 0 360 arc fill\r\n";
  print ps "$yscalew $width add 2.0 $sample_len mul
add $y moveto\r\n";
  print ps "$fsz 2 div $fsz 1 4 div mul neg rmoveto
($h[$i]) show\r\n";
  print ps "grestore\r\n";
  $y-=$fsz;
}
```

Drawing a legend (on the right of the chart) with the samples of curves (of different colors) and their titles.

```
for($i=1;$i<=@h-1;$i++)
{
  print ps "gsave $rgb_color[$i] setrgbcolor\r\n";
```

The loop for drawing the curves.

The graphical state is saved and the color of the curve is set from the above initialized table.

```
  $mvln="moveto";
  foreach $_ (@lines)
  {
    s/[\r\n]//g;
    @r=split/,/;
    if(/^([\d]+)(,([\d.]*))+$/)
    {
      if(defined($r[$i]) && $r[$i]!~/^\s*$/)
      {
        $x=$yscalew+($r[0]-$xmin)/($xmax-
$xmin)*$width;
        $y=$xscaleh+($r[$i]-$ymin)/($ymax-
$ymin)*$height;
        print ps "$x $y $mvln\r\n";
        $mvln="lineto";
      }
      else
      {
        $mvln="moveto";
      }
    }
  }
  print ps "stroke\r\n";
```

Drawing the curves.

Scanning the content of the input file, while considering only the i-th column (for the curve currently being drawn).

The first available point will invoke the moveto postscript command (preceded by the X Y coordinates of the point).

The following points will invoke the lineto postscript command.

If the curve is interrupted in the middle, then the first available point after the interruption must again invoke a moveto command.

Interruptions occur, if the data was not possible to retrieve from servers. We do not replace undefined values by zeroes.

At the end of the moveto and lineto sequences we stroke the curve.

```
  foreach $_ (@lines)
  {
    s/[\r\n]//g;
    @r=split/,/;
    if(/^([\d]+)(,([\d.]*))+$/)
    {
      if(defined($r[$i]) && $r[$i]!~/^\s*$/)
      {
        $x=$yscalew+($r[0]-$xmin)/($xmax-
$xmin)*$width;
        $y=$xscaleh+($r[$i]-$ymin)/($ymax-
$ymin)*$height;
```

Drawing the marker points of the curve.

Points are drawn by filling a circles of a $ptsz diameter.

`print ps "$x $y $ptsz 2 div 0 360 arc fill\r\n";` `}` `}` `}`	
`print ps "grestore\r\n";` `}`	The graphical state is restored and the end of the loop for drawing the curves and the points.
`print ps "showpage\r\n";` `close ps`	The last command of the Postscript page must be showpage.

4. The bash script monitoring the CPU loads on the remote computers and generating the CSV files

While running, the bash script generates and keeps updating a CSV file. The CSV file is stored in a cpu folder (do not forget to create this folder if you run the program on a new computer).

The script connects to all SIP servers simultaneously. While connected it runs on each server the top (cpu) program such that it displays 20 times, with intervals of 30 seconds, the cpu load of all ser processes. Each connection lasts a little bit less than 10 minutes. Connection to all servers are simultaneous, so all connections together also last about 10 minutes.

The values retrieved from each server during one connection session are averaged and one average CPU load is retrieved per server during 10 minutes. For several ser processes per server (8 processes usually), the retrieved average value corresponds to the total CPU consumption by all ser processes of an individual machine.

While the average CPU loads are retrieved from all servers with a periodicity of about 10 minutes, the graph is updated and is uploaded on the web server with a periodicity of about 20 minutes.

The bash script	**Comments**
`#!/bin/bash` `user=sona` `domain=youroute.net` `hosts="us1,ch1,fr1,fr2,fr3"` `topsamp=20` `topdelay=30` `loop=72` `upload=2` `delay=20` `noerror=1`	The parameter **`user`** contains the ssh username for the hosts being monitored. The parameter **`domain`** contains the internet domain name of hosts. The parameter **`hosts`** contains the comma separated list of hosts being monitored.

	The parameter **`topsamp`** contains the number of times the top program shall display the CPU usage per one ssh connection. The collected results are averaged. In this example an average of 20 values will be considered per ssh connection. The parameter **`topdelay`** is the interval in seconds between displays of the top program. Therefore the single ssh connection will last about (20-1)*30=570 seconds. **`loop`** contains the number of times the remote top program must be called (simultaneously on all machine). **`upload`** specifies the intervals at which the graph should be updated and uploaded to the web server. The parameter **`delay`** is an idle interval in the main loop (in case of an error in the body of the loop the program must not loop consuming the entire cpu resource). The duration of the entire bash script can be computed according the following formula: $loop * (($topsampl-1) * $topdelay + $delay) plus the time needed for communications: ssh (retrieval) and ftp (upload) communications.

`csv2eps=./b7.pl.txt`
 	
`convert=/usr/local/bin/convert`
`localdir=/root/folders/080212-portasip-ser-cpu`
`passfile=/root/files/070930-unappel-ftplogin.txt`
	`csv2eps` contains the name of the Perl script for converting the CSV files into a graphical format (see section 3).
	
	`passfile` is the name of the file containing the ftp password of the web server.

cd $localdir
csv=cpu/`date +%y%m%d.%H%M%S`-ser-cpu.csv
echo time,$hosts | tee $csv
eps=`dirname $csv`/`basename $csv .csv`.eps
gif=`dirname $csv`/`basename $csv .csv`.gif
	Computing the name of a CSV file (YYMMDD.HHMMSS-ser-cpu.csv).
	
	Creating the header line of the CSV file.
	
	Computing the names of EPS and GIF files.

ftp_cmd="set ftp:passive-mode no; cd htdocs/public/080210-ser-cpu; put $gif; bye"
ftp_usr="unappel,`cat $passfile`"
ftp_dst=www.unappel.ch
	ftp_cmd, **ftp_usr** (username and the password), and **ftp_dst** (the server name) strings will be used by the **lftp** tool when uploading the GIF files on the web server.

i=0
while [$i -lt $loop]
do
	Beginning the main loop for retrieving the average CPU load values.
	
	Each iteration of this loop results into one entry line in the CSV file.
	
	The CSV entry line contains the average CPU loads of all hosts.

(
    for h in `perl -e '@h=split /,/,"'$hosts'"; print "@h"'`
    do
	Here is the loop passing through all hosts.
	
	The Perl string of the example evaluates into the list of space separated hosts “us1 ch1 fr1 fr2 fr3”.
	
	The loop invokes asynchronous ssh processes. Therefore the connections are established simultaneously.
	
	Their outputs will be merged and parsed via a pipe to the next command (following this for loop and its body).

perl -e '{if(rand()<='$noerror'){exit 0}{exit 1}}'
      if [$? -ne 0]
      then
        h="$h-error"
      fi
      login="$user@$h.$domain"
	This part can be ignored.
	
	We introduce errors if the value of $noerror is less than 1.
	
	This can be needed to validate the comportment of the script when the hosts are not reachable.

ssh -q $login "top -b -d$topsamp -s$topdelay inf \| grep ' \\(ser\\\|COMMAND\\)$'" | perl -e	At each iteration of the current for loop we invoke an ssh connection in a asynchronous mode.

```
'$h="'$h'";
        while(<>)
        {
          s/^\s+//;
          @cols=split/\s+/;

          if(/ COMMAND$/)
          {
            for($i=0;$i<@cols;$i++)
            {
              if($cols[$i] eq "WCPU")
              {
                $wcpu_col=$i;
                $n++;
                break;
              }
            }
          }
          elsif(/ ser$/)
          {
            if(defined($wcpu_col))
            {
              $_=$cols[$wcpu_col];
              s/%//;
              $sum+=$_;
            }
          }

        }
```
	The connections are invoked simultaneously and not sequentially.
	
	The piping of the outputs to the next process groups all outputs. It waits until the output from each of the processes is completely received.
	
	The ssh connection does not require password. For this purpose we use public/private key authentication method (more about SSH login without password).
	
	The top program is called with –b option for batch mode (without screen control escape commands).
	
	The option –d is for specifying the number of displays, and the option –s for the delay (in seconds) between the displays.
	
	The grep permits to limit the ssh transmission (from the remote machine to the local machine).
	
	Only the lines containing the CPU loads of the ser processes and the lines containing the header line of the top program will be transmitted.
	
	The output of ssh is given to a Perl script which finds the column number containing the CPU load values.
	
	Note that the column is different on the different remote computers and this piece of script provides the required compatibility.
	
	The Perl script computes the number of occurrences of the header line and sums up the CPU loads.

`        if(defined($sum) && defined($n))`	When the output of the ssh connection is

```
    {
     printf "%s %s\n",$h,$sum/$n;
    }
```

over, the Perl script displays the average load of all ser processes.

```
   ' &
```

The ssh to Perl pipeline (one pipeline per each individual connection) is launched asynchronously in a background mode.

```
  done
 ) | perl -e '
```

The output of all asynchronous ssh processes will be merged into a single flow and will be pipelined into the next Perl script.

Thus, the next Perl scrip receives the average CPU loads of all consulted computers.

```
  while(<>)
  {
   @a=split/\s+/;
   $wcpu{$a[0]}+=$a[1];
  }
  print time();
  foreach $h
("us1","ch1","fr1","fr2","fr3")
  {
   printf ",%s",$wcpu{$h};
  }
  print "\n";
 ' | tee -a $csv
```

This Perl script receives the lines corresponding to each remote computer and stores the values in a list %wcpu (indexed by host names).

Then a single line, containing the current time and the values received from all computers (correspondingly sorted) is appended at the end of the CSV file.

```
 i=`expr $i + 1`
 if [ `expr $i % $upload` -eq 0 ]
 then
  $csv2eps $csv $eps
  if [ -f $eps ]
  then
   $convert $eps $gif
   lftp -e "$ftp_cmd" -u $ftp_usr
$ftp_dst 2>&1
  else
   echo "File $eps was not created"
  fi
 fi
```

Before the end of the iteration we check whether an updated graph must be uploaded.

If the graph must be uploaded, the CSV file is converted into an EPS file, which is further converted into a GIF file, and then uploaded to the web server using the lftp tool.

In case of CSV format errors (e.g. insufficient data values in the input file) the EPS file will not be created.

If the CSV file is not created, the script does not attempt to convert into a GIF file and to upload the GIF file on the web.

```
 sleep $delay
done
```

End of the main loop.

```
$csv2eps $csv $eps
if [ -f $eps ]
then
  $convert $eps $gif
  lftp -e "$ftp_cmd" -u $ftp_usr
$ftp_dst 2>&1
  if [ $? -eq 0 ]
  then
    rm $csv
    rm $eps
    rm $gif
  fi
else
  echo "File $eps was not created"
fi
```

At the end of the program we convert the final CSV file into a GIF file and we upload the final GIF file onto the web server.

The CSV, EPS, and GIF files are removed from the local computer.

[a6.sh.txt] version of the printout
[a7.sh.txt]
[a9.sh.txt] version updated on 2008-03-26

5. Version upgrades

In the upgrade from version [a6.sh.txt] to [a7.sh.txt] we removed a back (the list of hosts was hardcoded):

```
$ diff a6.sh.txt a7.sh.txt
6c6
< hosts="us1,ch1,fr1,fr2,fr3"
---
> hosts="us1,ch1,fr1,fr2,fr3,fr4"
89c89
<     foreach $h ("us1","ch1","fr1","fr2","fr3")
---
>     foreach $h (split/,/,"'$hosts'")
```

5.1. [update080326] using keep-alive control for simultaneous ssh sessions

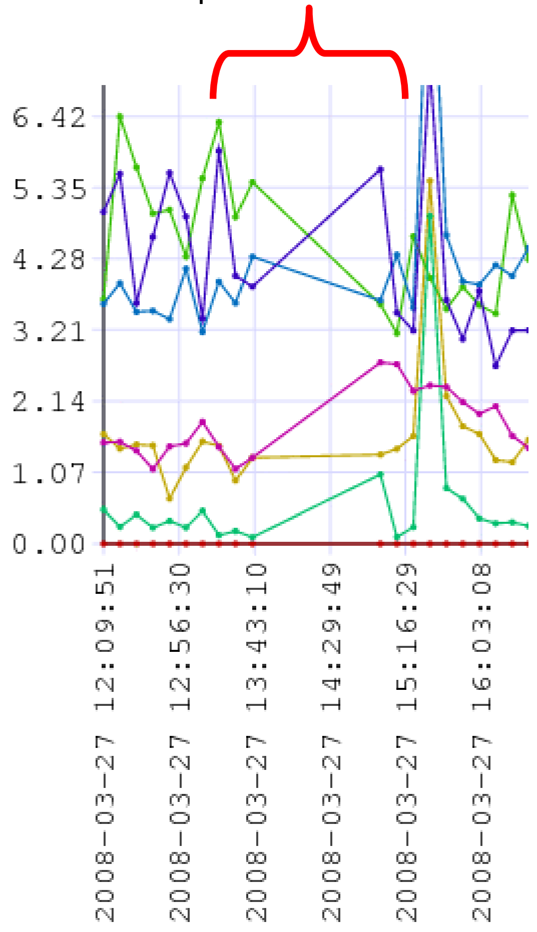
The upgrade from version [a7.sh.txt] to version [a9.sh.txt] takes care of dead servers. When one of the servers is dead, the entire script processing is delayed. The progress of all chart points is delayed for all hosts during the entire time of a suspended connection with the dead host. Deadlock-free ssh connection relies on keep-alive control messages aiming at detection of dead servers. The dead server is detected via a control channel without a need of user/application data exchanges:

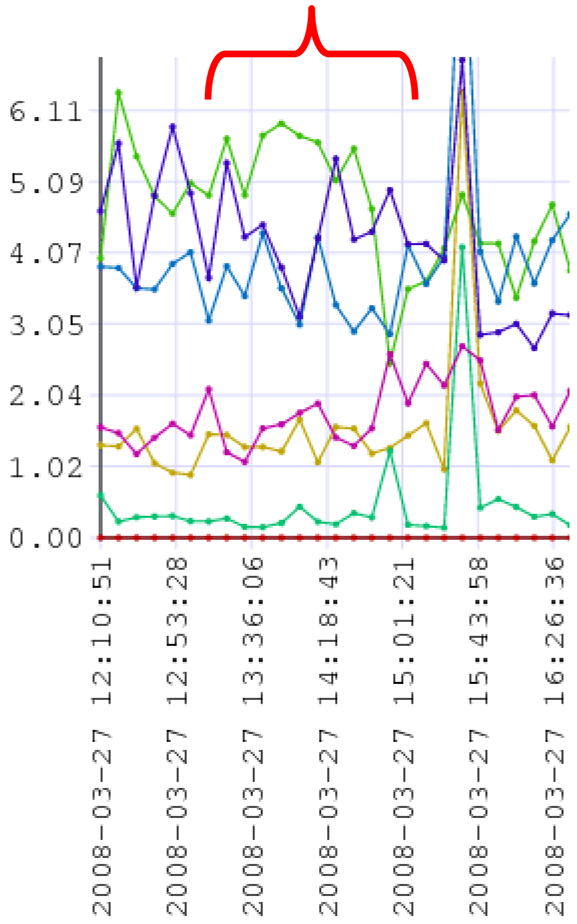
```
$ diff a7.sh.txt a9.sh.txt
6c6
```

```
< hosts="us1,ch1,fr1,fr2,fr3,fr4"
---
> hosts="us1,ch1,fr1,fr2,fr3,fr4,dk1"
47c47
<       ssh -q $login "top -b -d$topsamp -s$topdelay inf | grep '
\(ser\|COMMAND\)$'" | perl -e '$h="'$h'";
---
>       ssh -o ServerAliveInterval=5 -q $login "top -b -d$topsamp -
s$topdelay inf | grep ' \(ser\|COMMAND\)$'" | perl -e '$h="'$h'";
```

References for ssh option ServerAliveInterval:
http://4z.com/public/080326-ssh-keepalive/
http://www.unappel.ch/public/080326-ssh-keepalive/
http://switzernet.com/public/080326-ssh-keepalive/

The two charts below are generated for the same period of time by the old and new versions of the script. The first chart (on left), generated by old version [a7.sh.txt] demonstrates a case, when due to a bad (potentially lost) connection with one of the servers the drawing of all curves is delayed. The second chart (on right), generated by upgraded script [a9.sh.txt], shows a steady progress of all curves during the same period. The ssh session with affected server can terminate and the load value will be computed with as much samples as collected until the outage.





In general, when the ssh session is terminated in the middle of connection, the script computes the average CPU load with the data collected from the beginning of the session, but if the server stays unavailable until the next connection attempt scheduled at the following sampling period, then the CPU load curve of the affected server will interrupt.

6. Files and links

A short and efficient Perl tutorial (you do not need to know more for presented scripts): http://www.comp.leeds.ac.uk/Perl/, [cached]

A more complete Perl documentation: http://perldoc.perl.org/perlintro.html

The bluebook of postscript language: http://www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF, [cached]

Description of the EPS headers extending the PS language: http://www.tailrecursive.org/postscript/eps.html, [cached]

A scheme assigning colors to a range of scalar values: http://4z.com/People/emin-gabrielyan/public/050401-linkviews/

Description of the public and shared key ssh login process (ssh-keygen -t rsa): http://linuxproblem.org/art_9.html, [cached]

Imagemagick web site (the version on the Geneva server is installed from sources) http://www.imagemagick.org/script/install-source.php

This document: [htm], [doc], [pdf]

The bash and Perl scripts: [a9.sh.txt], [b7.pl.txt]

Mirrors: [ch1], [ch2], [us1]

* * *