Improved metrics collection and correlation for the CERN cloud storage test framework

September 2013

Author:
Carolina Lindqvist

Supervisors:
Maitane Zotes
Seppo Heikkila

CERN openlab Summer Student Report 2013

Abstract

Storage space is one of the most important ingredients that the European Organization for Nuclear Research (CERN) needs for its experiments and operation. Part of the Data & Storage Services (IT-DSS) group's work at CERN is focused on testing and evaluating the cloud storage system that is provided by the openlab partner Huawei, Huawei Universal Disk Storage System (UDS). As a whole, the system consists of both software and hardware.

The objective of the Huawei-CERN partnership is to investigate the performance of the cloud storage system. Among the interesting questions are the system's scalability, reliability and ability to store and retrieve files. During the tests, possible bugs and malfunctions can be discovered and corrected. Different versions of the storage software that runs inside the storage system can also be compared to each other.

The nature of testing and benchmarking a storage system gives rise to several small tasks that can be done during a short summer internship. In order to test the storage system a test framework developed by the DSS group is used. The framework consists of various types of file transfer tests, client and server monitoring programs and log file analysis programs. Part of the work done was additions to the existing framework and part was developing new tools. Metrics collection was the central theme. Metrics are to be understood as system statistics, such as memory consumption or processor usage.

Memory usage and disk reads/writes were added to the existing client real-time monitoring framework. CPU and memory usage, network traffic (bytes received/sent) and the number of processes running are collected from a client computer before and after a daily test. Two other additions are visualization for storage system log files, as well as a new monitoring tool for the storage system. This report is divided into parts describing each part of the framework that was improved or added, the problem and the final solution. A short description of the code and the architecture are also included.

Table of Contents

1 Introduction

The complete Huawei Universal Disk Storage (UDS) system occupies less than three server racks. It has 768 TB of storage space that is divided over 384 storage nodes (SD) and controlled by seven controller nodes as pictured in Fig. 1. Both software and hardware are part of this storage system.

Testing is done by uploading and downloading files of different sizes to the storage system in order to stress it. Small file sizes can be used to test how the system handles metadata. The upload and download tests are performed from several client machines that stress the storage system with as many file transfers as possible. The client machines are running Scientific Linux CERN 6 (SLC6). Scalability is tested by having multiple threads performing file transfers.

During the tests the clients are also monitored to ensure that they themselves are not a bottleneck, but instead stressing the storage system to perform at its full capability. After the tests, log files are collected from the storage system and analysed in order to understand the behaviour of the storage system.

The main results of my work are added metrics to the client side monitoring, a visual representation of the server side logs as graphs, as well as an implementation of storage node monitoring. In the following, each of these topics are described separately by presenting the wished feature and the outcome of the work.

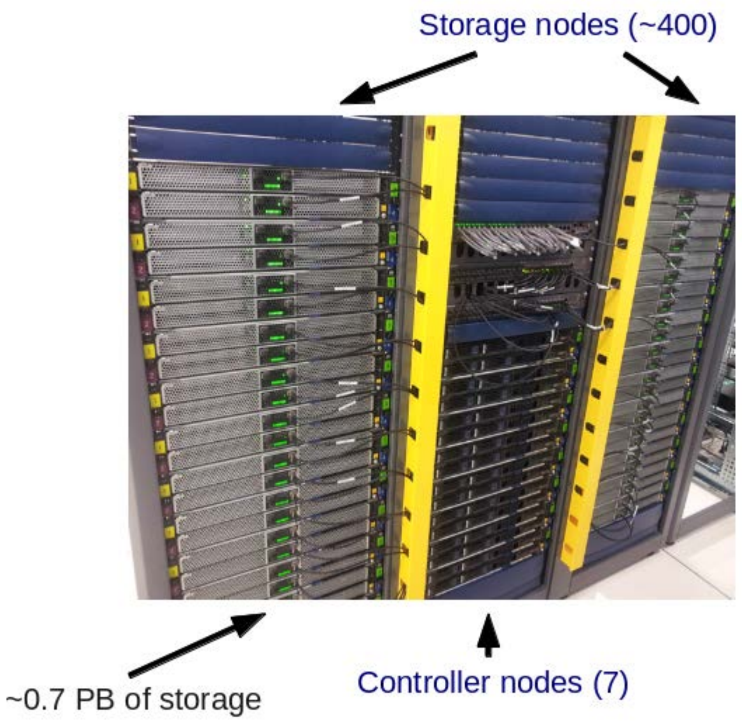


Figure 1. The Huawei UDS storage system.

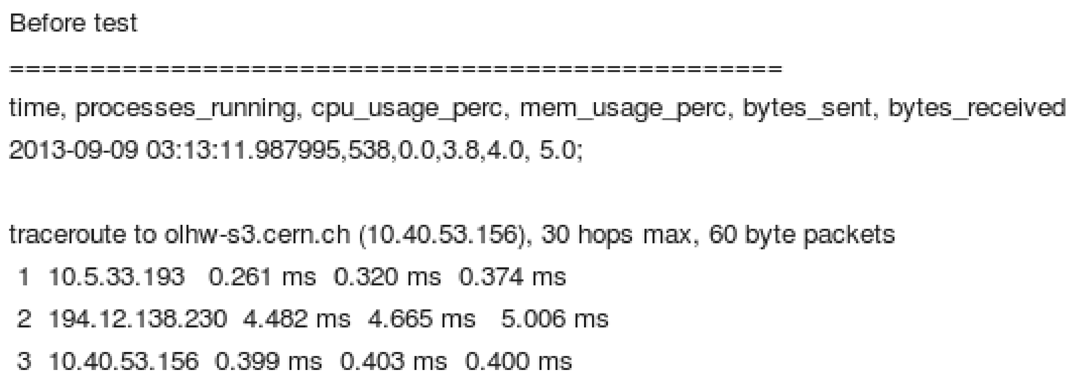
2 Metrics collection for the morning test

The morning test consists of a single client machine that downloads files from the storage system using a single thread and then 100 threads. A download is done five times during one test run and an average is calculated. This whole test is performed each day as a cron job in order to ensure that the storage system works properly. At the end of the test, a report containing the results is automatically dispatched by e-mail. This report previously only contained the amount of time the downloads needed to finish, the average time, the amount of threads and a timestamp. The additions are shown in Fig. 2.

In order to improve the test, the following metrics were added (measured from the client):

- CPU and memory usage in percentage.
- The amount of processes running.
- The amount of data (measured in bytes) that the client sends and receives during an interval of ten seconds.
- The IP network route from the client to the management node.

The route from the client to a management node was added to display any significant changes to the network. This is collected from the *traceroute* program. These additional metrics are measured before and after the whole test run. In this way, it is possible to detect whether major differences in the storage system performance can be explained by external load on the client or disturbances in the network, that has caused the client not to stress the storage system enough.



```
Before test
================================================
time, processes_running, cpu_usage_perc, mem_usage_perc, bytes_sent, bytes_received
2013-09-09 03:13:11.987995,538,0.0,3.8,4.0, 5.0;

traceroute to olhw-s3.cern.ch (10.40.53.156), 30 hops max, 60 byte packets
 1  10.5.33.193   0.261 ms  0.320 ms  0.374 ms
 2  194.12.138.230  4.482 ms  4.665 ms   5.006 ms
 3  10.40.53.156  0.399 ms  0.403 ms  0.400 ms
```

Figure 2. An excerpt with metrics from a sample morning test report.

The existing morning test script is written in Python and was extended using two external python modules. The psutil v.1.0.1 module[1] was used for metrics collection. The pexpect v.2.3 module[2] is used to spawn a process that performs the traceroute. The script is

[1] http://code.google.com/p/psutil/

[2] http://www.noah.org/wiki/pexpect

intended to run on SLC6 using Python v. 2.6. Lastly, these metrics measurements were added to an existing bash script that starts the test and sends the final report mail. In addition to the mail, all measurements are stored in a log file.

3 Additional metrics for client monitoring

The client monitoring is done using a previously developed software that measures the CPU and network usage. The software reads files from the /proc file system that can be found in a Linux file system. These files contain information about the operation system, for example, network statistics can be found from the file /proc/net/dev. This monitoring software was developed in C++. The ROOT software was used for drawing graphs and histograms of the measurement data. All measurements are done in near real-time. This software consists of clients that are executed on the monitored computers and a master node that collects measurements that are sent from each client and finally assembles a ROOT file that contains the readings and histograms.

Measured metrics are configurable in a bash script. One test run can contain file uploads and downloads, done using different amounts of threads. The client C++ code uses a python wrapper to run the actual test code that is written in Python.

The additional metrics that were added to the client monitoring were the memory usage in percent and disk IO metrics. A screenshot from the monitoring during a test can be found below in Fig. 3. These metrics can reveal any significant changes in the clients during file upload and download. The CPU and memory metrics measured from the client should stay as stable as possible. The measured disk reads and writes as well as the network traffic should correspond to the traffic that the test run generates, for example file downloads.

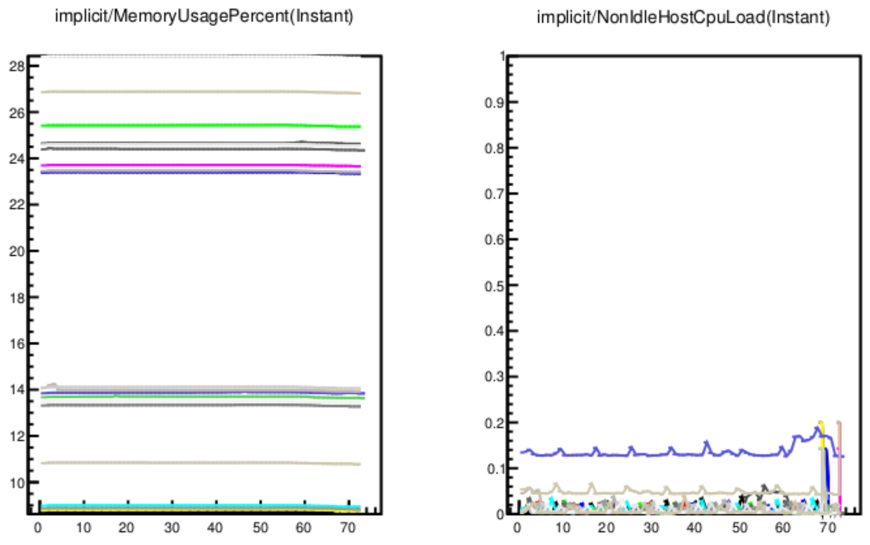


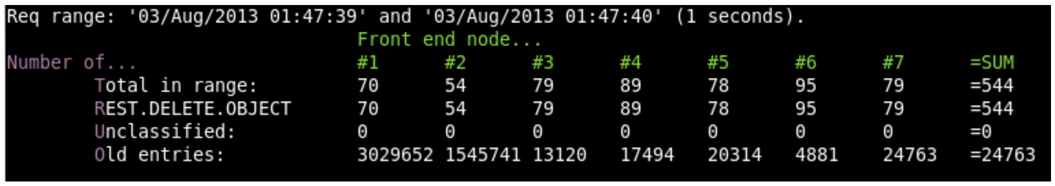
Figure 3. Screenshot from the metrics added to the monitoring software.

4 Visualizing log files

Log files are continuously collected inside the storage system. The log files contain timestamped entries of the file operations, errors or status messages that the storage software has produced. These log files can then be parsed into more readable results and reports. Analyzing these log files is an important part of understanding the behavior of the storage system.

Two different types of log files are collected on the storage system servers ; access logs and logs from the storage software. The access log files contain information about the file operations that have been performed inside the storage system, for example a GET request or a DELETE request for a file. Each operation is logged with the precision of a millisecond. The information contained in these logs can be correlated with the tests that are run against the storage system, ie. in order to verify that an attempt to download files from the storage system creates GET requests.

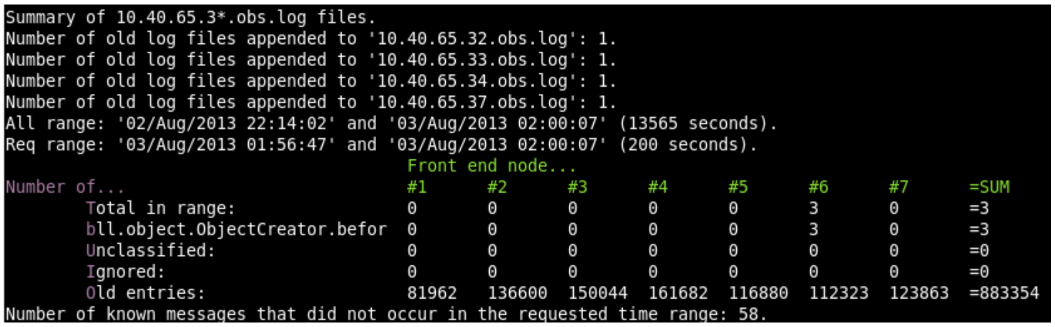
The log file parsing that previously existed enabled the user to choose an interval during which certain file operations were extracted from the log files and the amount of each operation that had been registered were presented in a table. A sample excerpt from this table is illustrated in Fig. 4.



```
Req range: '03/Aug/2013 01:47:39' and '03/Aug/2013 01:47:40' (1 seconds).
                                Front end node...
Number of...                    #1      #2      #3      #4      #5      #6      #7      =SUM
        Total in range:         70      54      79      89      78      95      79      =544
        REST.DELETE.OBJECT      70      54      79      89      78      95      79      =544
        Unclassified:           0       0       0       0       0       0       0       =0
        Old entries:            3029652 1545741 13120   17494   20314   4881    24763   =24763
```

Figure 4. Previously existing log file parsing.

The second kind of log files contains entries from the storage system software. Any kind of error or event that is logged can be found in these files. When the requested range to be analysed is longer than the contents of the most recent log file, the parser appends log files that are older as needed. Before, appending was only done for access log files. The first improvement was to add the ability to append old log files also from the storage system software logs. In Fig. 5 the appending and analysis of software logs is visible.



```
Summary of 10.40.65.3*.obs.log files.
Number of old log files appended to '10.40.65.32.obs.log': 1.
Number of old log files appended to '10.40.65.33.obs.log': 1.
Number of old log files appended to '10.40.65.34.obs.log': 1.
Number of old log files appended to '10.40.65.37.obs.log': 1.
All range: '02/Aug/2013 22:14:02' and '03/Aug/2013 02:00:07' (13565 seconds).
Req range: '03/Aug/2013 01:56:47' and '03/Aug/2013 02:00:07' (200 seconds).
                                    Front end node...
Number of...                        #1      #2      #3      #4      #5      #6      #7      =SUM
        Total in range:             0       0       0       0       0       3       0       =3
        bll.object.ObjectCreator.befor 0    0       0       0       0       3       0       =3
        Unclassified:               0       0       0       0       0       0       0       =0
        Ignored:                    0       0       0       0       0       0       0       =0
        Old entries:                81962   136600  150044  161682  116880  112323  123863  =883354
Number of known messages that did not occur in the requested time range: 58.
```

Figure 5. Appending and analysing log files from the storage system software.

The main new feature that was added is the possibility to draw a graph that shows the distibution of a chosen file operation during the given interval based on log file entries. The graph can be plotted separetely for each frontend node or as a sum of all nodes. An example can be found as Fig. 6. Unit tests were also added in order to ensure that the correct entries are read from the log files.

.

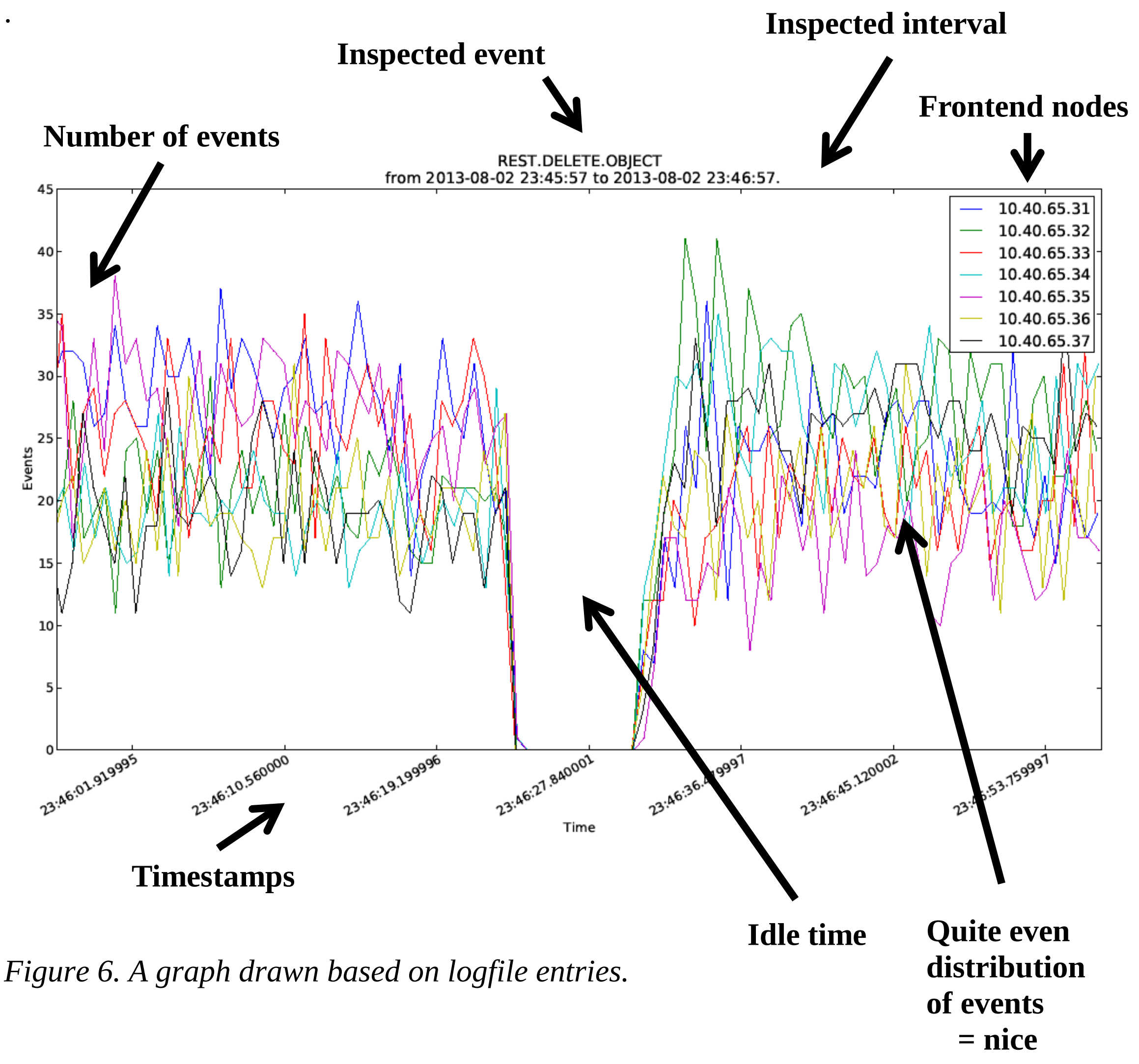


Figure 6. A graph drawn based on logfile entries.

In Fig. 6 a sample analyzed log file is presented. Each graph represents a frontend node that handles the requested file operations, for example a DELETE request that removes a file. The operation is shown as the title of the graph with the requested interval that is inspected below it. On the y-axis the amount of events that have occured are presented. Each peak on the curve represents the amount of file operations that have been binned into wanted intervals, such as one second.

The visualization can assist in detecting malfunctions, for example the large gap that is seen in Fig. 6 can be attributed to an interval between test runs. If this gap had not been expected, it would have been fairly easy to spot that something had happened inside the storage system. It can also be seen that the load on the frontend nodes is distributed quite evenly. If one of the nodes would have registered few events in its log, while another node would have exceptionally many log entries, this could easily be spotted in a graph.

5 Server side monitoring

To ensure that the storage nodes (SN) operate correctly, it would be interesting to monitor their use of resources similarly as the clients are monitored during a test. Due to the restrictions listed below it is not possible to use the same software that is used to monitor the clients. The following limitations apply to the scenario of monitoring the 384 storage nodes. Nothing should be installed on the storage nodes; otherwise this could conflict with the performance of the storage system. This excludes the reuse of the monitoring software that is used with the clients, since it would require to be compiled for the storage nodes. The nodes can only be accessed over a secure connection through one of the management nodes. Information should be received in near real-time.

As a solution, since every SN runs a tailored Linux distribution, system statistics are only read from the /proc filesystem and obtained from tools already available, for example 'iostat'. The SNs are accessed from a desktop that is running the monitoring software, by a bash script 1) that loads a bash script for monitoring into a variable, 2) sent through the management node (MN) and 3) finally expanded and executed on the SN. This is done over a secure connection. The monitoring script collects the requested data with configurable intervals, for example one second. The SOD monitoring process is illustrated in Fig. 7.

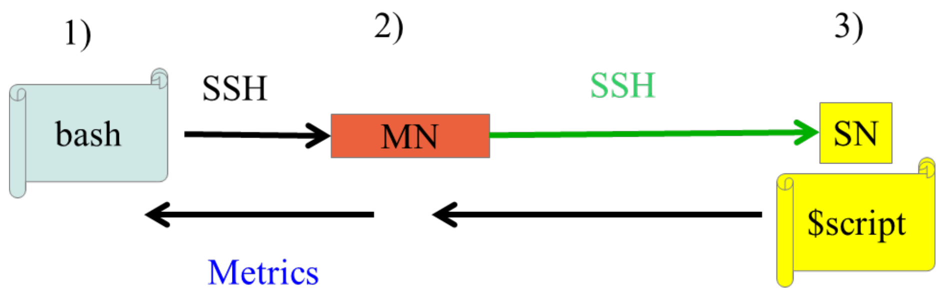


Figure 7. The concept of storage node monitoring.

Every reading is then sent back to the bash script 1) that stores the readings into timestamped files named according to the SN's IP-address. These log files can then be plotted using a Python script that parses the values from the file and draws a plot using the MatPlotLib python module. Currently only the CPU metric is available.

The script is configurable to monitor nodes listed in a configuration file. The duration of the monitoring time is also configurable. A sample reading from SOD nodes is shown in Fig. 8. A sample reading from the lxbsp nodes that are used as clients during upload and download tests is shown in Fig. 9.

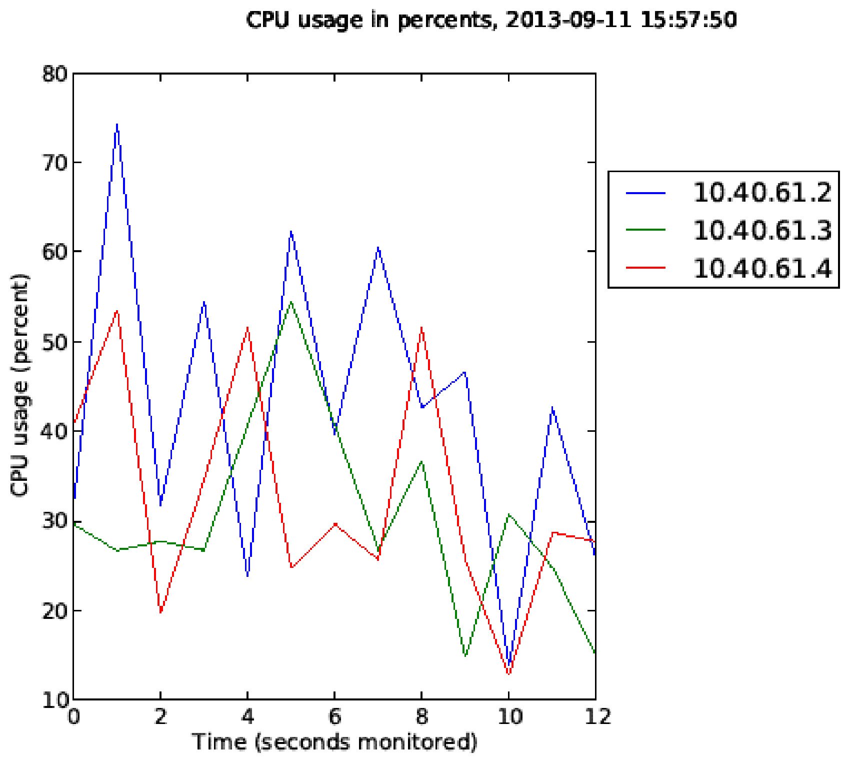


Figure 8. CPU usage in SOD nodes.

One can easily notice the difference in activity between nodes, for example the lxbsp20b27 node that shows a higher activity than the other nodes in Fig. 9. As a contrast, the SOD nodes show more fluctuations and a high activity in the sample reading presented in Fig. 8.

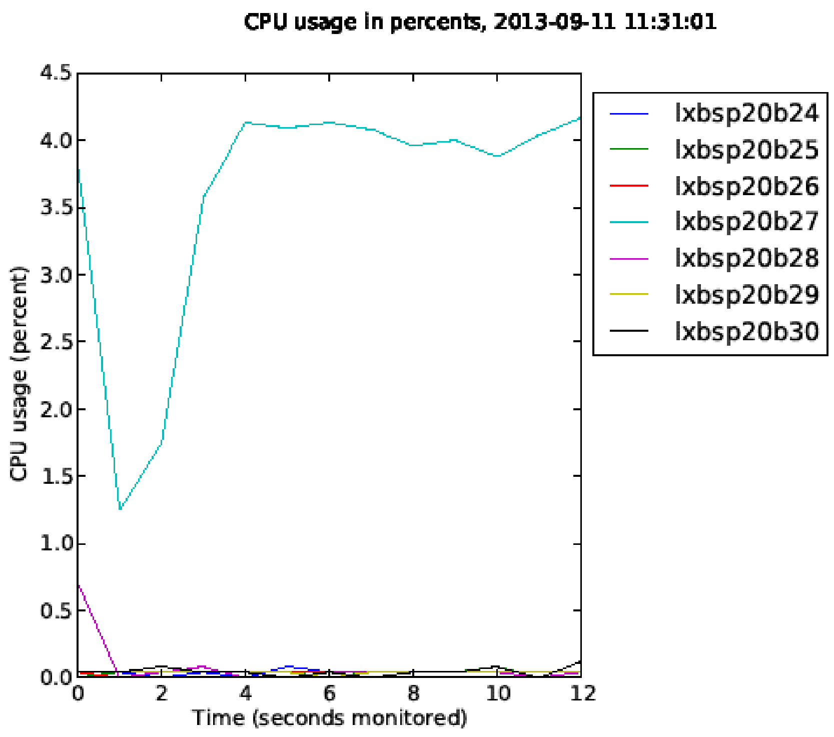


Figure 9. CPU usage in lxbsp nodes.

6 Conclusions and future work

The results of this work are new metrics that have been made available and new tools that can be used for analysing these metrics. For example, CPU and memory usage, the number of processes running and network send/receive statistics (in bytes) are now collected from a client before and after a daily test that it runs. These metrics can identify the client as a possible bottleneck in the test setup. The visualisations of the server side log files that have been made available are easier to read and help to understand the status of the storage system at a certain point in time. Monitoring the storage nodes (SODs) was not possible before, but now a new tool for this purpose exists. Currently, CPU and memory usage, HDD read/writes and network traffic can be stored into logfiles and the CPU metric can be drawn as a plot from such a log file.

In the future, the client monitoring system could still be extended to support additional metrics, for example notifications when certain threshold values are reached. More unit tests could be added to the test framework in general, in order to ensure that the code is not broken during development. The plot functionality from the server side monitoring could easily be extended to support multiple metrics, in addition to the CPU metric. The original intention was also to plot the server side monitoring readings in real time, similarily to what is done on the client side. This is also a possible extension to be done.

7 References

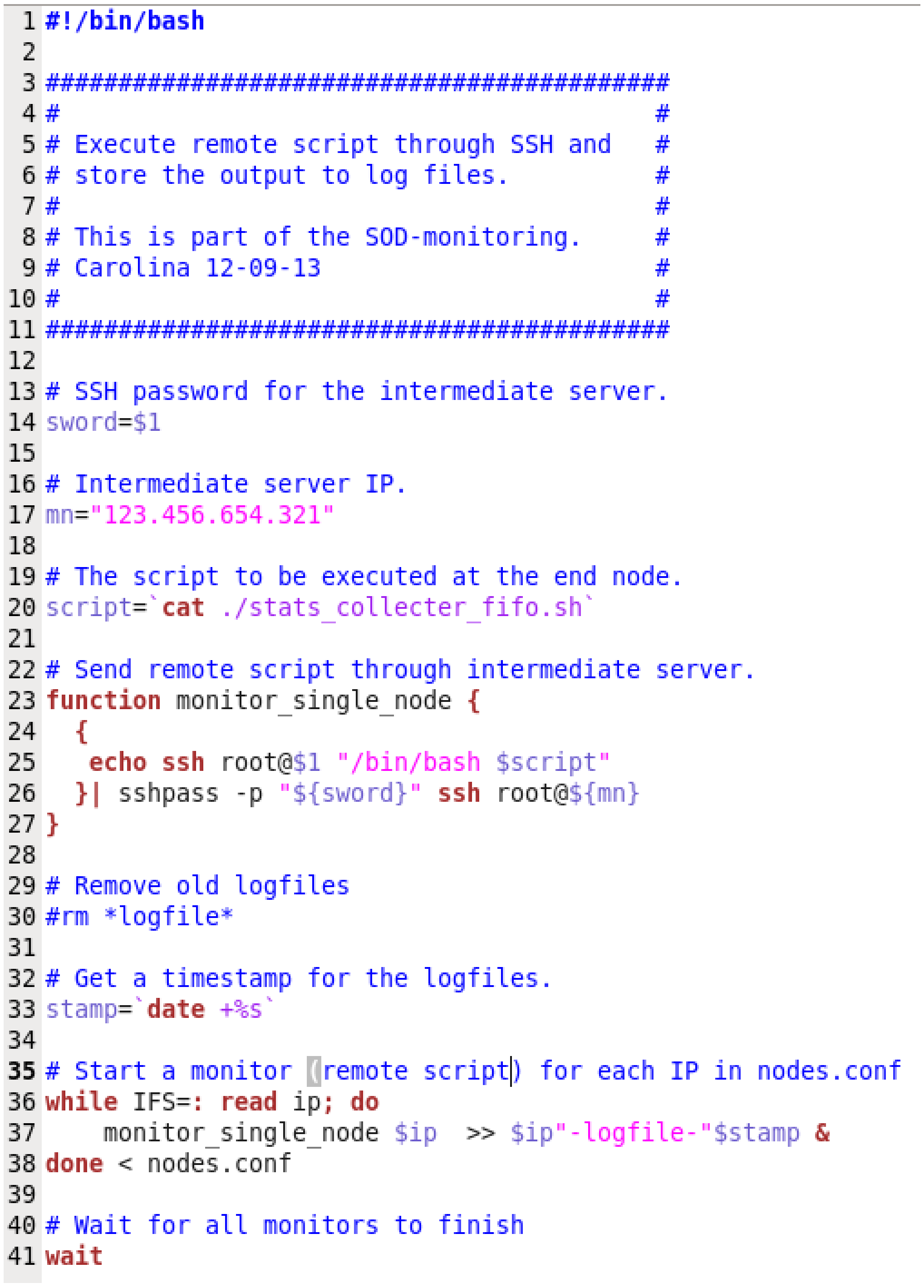
Python libraries :

Metrics collection: psutil-1.0.1 (http://code.google.com/p/psutil/)

Spawning processes: pexpect-2.3 (http://www.noah.org/wiki/pexpect)

8 Appendix 1

Sample bash script for executing another script remotely over SSH. It is used with the suitable parameters to send a script through the MN to be executed at the SNs listed in a file called ‘nodes.conf’ as described in Fig. 7.



```bash
#!/bin/bash

###########################################
#                                         #
# Execute remote script through SSH and   #
# store the output to log files.          #
#                                         #
# This is part of the SOD-monitoring.     #
# Carolina 12-09-13                       #
#                                         #
###########################################

# SSH password for the intermediate server.
sword=$1

# Intermediate server IP.
mn="123.456.654.321"

# The script to be executed at the end node.
script=`cat ./stats_collecter_fifo.sh`

# Send remote script through intermediate server.
function monitor_single_node {
  {
   echo ssh root@$1 "/bin/bash $script"
  }| sshpass -p "${sword}" ssh root@${mn}
}

# Remove old logfiles
#rm *logfile*

# Get a timestamp for the logfiles.
stamp=`date +%s`

# Start a monitor (remote script) for each IP in nodes.conf
while IFS=: read ip; do
    monitor_single_node $ip  >> $ip"-logfile-"$stamp &
done < nodes.conf

# Wait for all monitors to finish
wait
```

Appendix 1

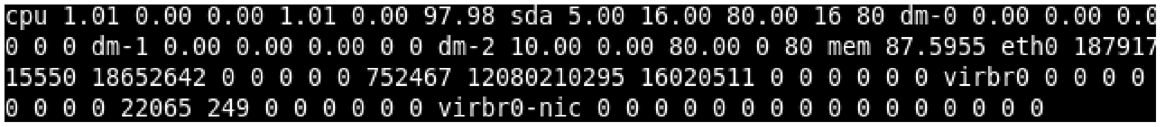
9 Appendix 2

Sample monitoring script that is executed on the SOD nodes to collect CPU, memory, network usage and disk usage metrics. Additional metrics can be added to row 31.

```bash
 1 #!/bin/bash
 2 # Box monitoring
 3
 4 # For how long we will monitor ($1).
 5 duration=15
 6
 7 # With what interval reports are sent ($2).
 8 interval=1
 9
10 # Prepare memory and network metrics
11 MEM="free | grep Mem | awk '{print \"mem \" \$3/\$2"'*'"100.0}'"
12 NET="cat /proc/net/dev | awk '/eth/,0' | tr ':' ' '"
13
14 # Create fifo and redirect iostat output there.
15 fifo="/tmp/temp"
16 mkfifo $fifo
17 $(iostat 1 $duration >"$fifo") &
18
19 first=1
20 printout=""
21
22 # Read the iostat output from the fifo.
23 while read line; do
24   if [ "$first" != "1" ]; then
25     case "$line" in
26      *Device*)
27      # Skip device line [Device: tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn]
28        ;;
29      *avg-cpu*)
30      # Skip the avg-cpu line [avg-cpu:  %user   %nice %system %iowait  %steal   %idle] & get new reading
31          echo -n "cpu" $printout $(eval $MEM) $(eval $NET) #"time" $(date +%s)
32          echo
33          printout=""
34        ;;
35      *)
36        printout="$printout"" ""$line"
37        ;;
38     esac
39   fi
40
41   # Only to skip the first row with system information from iostat.
42   case "$line" in
43     *avg-cpu*)
44       first=0
45       ;;
46   esac
47   #echo $printout
48 done <"$fifo"
49
50 # Remove the fifo.
51 rm "$fifo"
```

Appendix 2. SOD monitoring script.

The script produces the following output for example, from which metrics can be parsed and plotted. The output is stored in log files:



```
cpu 1.01 0.00 0.00 1.01 0.00 97.98 sda 5.00 16.00 80.00 16 80 dm-0 0.00 0.00 0.0
0 0 0 dm-1 0.00 0.00 0.00 0 0 dm-2 10.00 0.00 80.00 0 80 mem 87.5955 eth0 187917
15550 18652642 0 0 0 0 0 752467 12080210295 16020511 0 0 0 0 0 0 virbr0 0 0 0 0
0 0 0 0 22065 249 0 0 0 0 0 0 virbr0-nic 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Appendix 2.1 Sample monitoring script output.

The script output depends on the hardware, for example network interface cards and hard disk drives, all of which are listed.