GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Monitoring System Performance

Lukas Steinegger

MatrNr: 27118007

Supervisor: Marcus Merz

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2023

Abstract

Collecting system performance metrics is important to grasp the current state and health of the underlying system. By determining where performance loss comes from, we can improve our system in a targeted manner.

We look at three different tools, namely cc-metric-collector, Performance CoPilot, and Telegraf. Furthermore, we discuss the Inlfux one-line protocol utilized by the InfluxDB. We were also able to implement our own tool using Python, and go into detail about major design decisions and what motivates them. The source of the system metrics is the */proc* directory under Linux. Due to the scope, we only explain a small set of selected files and values. The discussed files are */proc/stat*, */proc/<PID>/stat*, and */proc/meminfo*. We close with a discussion about drawbacks, limitations, and possible future work.

Contents

List of Listings

List of Abbreviations

PCP Performance Co-Pilot

DB Database

JSON JavaScript Object Notation

OS Operating System

CPU Central Processing Unit

PID Process ID

REST Representational State Transfer

API Application Programming Interface

PMCD Performance Metrics Collection Daemon

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

1 Motivation

As system administrators, we want to detect outages in our system. Ideally, we want to prevent them, and in the worst case, we want to know if the system is currently down so that we can take action to get the system up and running again. All of that should happen in a reasonable amount of time. Furthermore, if the system is going to crash, we want to know that in advance, so we can take precautions (e.g., buy more Hardware, throttle traffic). If the system is down, we want to know about that the moment it goes down. System Monitoring can be as easy as calling `ps` in the command line and increase up to a sophisticated monitoring architecture. However. we are not only interested in the worst case but want to ensure that all users have a pleasant experience with our system. The system should therefore be responsive and give responses in an appropriate time. Monitoring is important to achieve a performant system. Knowing where the system is not fully productive can help us to make significant improvements.

2 System Performance Monitoring

To be able to monitor the system, we need to collect some meaningful data points so that we can derive the performance of the system from these data points. Furthermore, we need to identify what parts of the system we want to monitor. Some generally interesting features include the CPU, memory, storage, and network metrics. But it is also important to know how the user perceives the system. Thus, we should, for example, look into the response time and/or latency of the system. For the scope of this report, we focus on CPU and memory usage.

Another important point is the differentiation between the performance of the whole system, especially hardware, OS, and the performance of the applications. In other words, even when the hardware works fine, the software still needs to be monitored.

After we have determined what we want to monitor and through what metrics, we need actual processes to automate the sensing step. For this purpose, we have small programs called performance agents running in the background of the nodes. The task of those programs is to collect the desired metrics and send them to a central place for storage and further analysis.

3 Performance Agents

In this report, we take a closer look at three implementations. Additionally, we propose an approach in our own implementations.

- cc-metric-collector [1]
- Performance Co-Pilot [2]
- Telegraf [3]

[1] `https://github.com/ClusterCockpit/cc-metric-collector`
[2] `https://pcp.io/`
[3] `https://www.influxdata.com/time-series-platform/telegraf/`

- Our own implementation[4]

3.1 cc-metric-collector

The cc-metric-collector is part of the ClusterCockpit suite, along with the cc-metric-store and the cc-backend. This tool is configured through a configuration file. By default, the "config.json" file is in the current directory. The configuration file can also be specified via the ––configure option.

This tool is set up in multiple parts.

1. sinks: In the *sinks* section, it is specified where the collected data is sent to. Among others, it can be selected from *stdout*, *http* and *influxdb*.

2. collectors: Here it can be specified what to collect and where to collect it from. The tool allows a lot of choices. For us, some important ones were: *cpustat*, *memstat*, *iostat*, which collect the data from the
proc directory under linux. Note that the tool is also able to collect metrics from other sources, e.g., *likwid* [5]

3. receivers: With the receivers it can be specified that the tools forward information to the defined sinks. We did not look further into this feature of the tool.

4. router: With the *router* entries the tool allows us to add or remove tags to each measuring point. Those can be specified via conditions in the JavaScript Object Notation (JSON) format in the config file.

A drawback of this tool is that its documentation is inaccurate. For example, the *interval* value was shown to be able to accept an integer value, but it only accepts strings with the interval given in value concatenated with the unit. Furthermore, such errors were not caught by the implementation, and the user received a non-filtered error output from the Go[6] implementation.

Apart from that, the tool was easy to install by following the instructions from their GitHub repository. Likewise, it was easy to configure the tool as wanted and to run an executable. First and foremost, the cc-metric-collector collects system metrics and does not collect process-specific data.

3.2 Performance Co-Pilot

The Performance Co-Pilot (PCP) comes with a wide range of tools, a description of some tools is listed below.

- pmstat gives a high level overview of the collected system metrics. It outputs a single line of a predefined set of values in a periodic way (this can be specified by the *-t* parameter.). In 3.2 we shown an example output of the program.

- pmval outputs a single metric, also in a periodic way. The metric to be displayed is selected as an argument (e.g. `pmval proc.nprocs`). The available metrics can be viewed using the pminfo utility.

[4] `https://gitlab.gwdg.de/lukas.steinegger/performance-agent`
[5] `https://github.com/RRZE-HPC`
[6] `https://go.dev/`

```
user@node:~$ pmstat
@ Mon Mar 27 17:41:14 2023
 loadavg                     memory       swap         io     system         cpu
   1 min   swpd   free   buff  cache   pi   po   bi   bo   in   cs  us  sy  id
    0.87  54784  1286m   3896 887668    0    0    0  926  955 1624   6   1  92
    0.80  54784  1274m   3896 899184    0    0    0 1001  837 1482   5   1  94
    0.73  54784  1279m   3896 887588    0    0    0  788  751 1213   4   1  95
    0.75  54784  1071m   3896 936064    0    0 4579 4323 3471 8369  28   8  64
    0.93  54784 725936   3896  1090m    2    0 1050 6513 5754  16K  54  13  33
    1.02  54784 769588   3896  1026m    0    0    0 3114 3154 9734  18   6  77
    1.02  54784 815224   3896   990m    0    0    0 1863 1253 2397   6   2  93
    1.02  54784 848744   3896 986876    0    0    0 1102  891 1452   7   1  93
```

Listing 1: Example output of the *pmstat* program

- pminfo can be used to display and to search for specific metrics. `pminfo proc` shows all available metrics with the prefix *proc*. The *-T* option is very useful because it gives a small description of the metric.
- pmstore can be used to set some metrics by hand. This can be useful when a metric needs to be reset (e.g. a counter).

PCP comes with a Performance Metrics Collection Daemon (PMCD) which runs on each host that needs to be monitored. The tool is configured by the *pmcd.conf*. The Daemon loads the configuration file through the `$PCP_PMCDCONF_PATH` environment variable. *pmcd.conf* has two parts; the first part configures what metrics are collected and the second part controls the access to the metrics.

A major drawback is that the apt package did not install all necessary dependencies, so that all programs function as intented. Furthermore, some tools did not work and only threw an unhandled Python error that said that the 'pcp' module was not found. In addition, the pmcd daemon could not be started with the system we were using. We tried to install the tools in a docker container using Fedora, but there we run into other issues. Since we could not resolve the errors in a reasonable amount of time, we kept the preview of the tool at this scope.

3.3 Telegraf

This tool is part of InfluxDB 3.3.1. It is configured through a configuration file which can be specified by the *–config* option. A default configuration file can be created by running `telegraf config > telegraf.config`.

After the setup of the tool, it is run using `telegraf --config telegraf.config`. It then collects the data and forwards them to the specified source, and if specified it can also output them using the Inlux one-line protocol to stdout. We show such an sample configuration in 2.

The tool is feature-rich, as it is able to read metrics from a wide range of sources. For example, it has built-in support to read metrics from {CockroachDB, MariaDB, Microsoft SQL Server, MySQL, PostgreSQL, SQLite}[7].

The other tools we have examined are not able to collect metrics directly from Database (DB) instances. With a simple query[8] over the default configuration file, Telegraf can read

[7] `https://github.com/influxdata/telegraf/blob/master/docs/SQL_DRIVERS_INPUT.md`

[8] telegraf config | grep -v "##" | grep "inputs\.[^\\.]*]]"

```
[agent]
  interval = "8s"
  round_interval = true
  collection_jitter = "0s"
  flush_interval = "8s"
  flush_jitter = "0s"

[[outputs.file]]
  files = ["stdout"]
  data_format = "influx"

[[inputs.cpu]]
  percpu = true
  totalcpu = true
  collect_cpu_time = false

[[inputs.processes]]
```

Listing 2: Small example configuration file for Telegraf.

metrics from around 200 inputs. It can also handle many destinations for the data. Just to name a few, it can load data via WebSockets, MongoDB[9], Apache Kafka[10], MQTT[11], and InfluxDB[12].

A sample configuration file could look like the following.

Overall, Telegraf is a powerful tool with many configuration possibilities, and when set up, it is easy to use. It works nicely together with InfluxDB. The documentation is well and so far we didn't encounter any errors. An additional benefit is the big community around Telegraf and InfluxDB, which helps to solve problems fast. By following the instruction on the website, the installation also worked without any problems. Telegraf can be used to collect system specific as well as process-specific data.

3.3.1 InfluxDB

This section shortly covers InfluxDB, since it is used at the GWDG and seems to be a widely used DB for performance metrics. Note the Influx-one-line protocol[13]: it is adapted by Telegraf and also by the cc-collector. PCP doesn't directly support this protocol. The InfluxDB can load sample points in this format directly into the DB instance. The syntax is as follows.

```
measurement,tags fields timestamp?
measurement := name of the measurement e.g. cpu_utility
tags := comma separated list of tag_name=value pairs
fields := comma separated list field_name=value pairs
timestamp (is optional, but often used) local timestamp in UTC,
```

[9] https://www.mongodb.com/
[10] https://kafka.apache.org/
[11] https://mqtt.org/
[12] https://www.influxdata.com/
[13] https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/

```
cpu,host=node,cpu=cpu user=0.073,system=0.014,idle=0.909 1679927146
cpu,host=node,cpu=cpu0 user=0.1725,system=0.01,idle=0.8175 1679927146
cpu,host=node,cpu=cpu1 user=0.1,system=0.0275,idle=0.865 1679927146
cpu,host=node,cpu=cpu2 user=0.010,system=0.007,idle=0.982 1679927146
cpu,host=node,cpu=cpu3 user=0.010,system=0.010,idle=0.972 1679927146
```

Listing 3: Shortened example output from our implementation of the Influx one-line protocol.

```
if not specified it is set by the InfluxDB instance
```

The *measurement* and *tags* lists are separated by a comma, but the *tags* list and the *fields* list are separated by a whitespace. The timestamp is also separated by a whitespace. The tag names and field names as well as their corresponding values can not contain whitespaces or commas, and they need to be encapsulated by quotation marks. The datatypes of values can be floats, integers, strings, and booleans. Floats are represented by a string of digits with a possible decimal point. Integers are only a string of digits with a suffix '*i*'.

Strings are a sequence of characters, and if they contain a comma or a white space, they need to be double-quoted. In the case of Booleans, true is represented with {T, t, True, true, TRUE} and false is represented with {F, f, False, false, FALSE}.

Tags are used to describe the data point, while fields are used for the actual measurement. For example, the hostname would be specified in the tags list. The CPU frequency, on the other hand, would be listed in the fields section. In 3.3.1 we show examples of the one-line protocol.

4 Own Implementation

Throughout the course, we were able to implement our own version of a performance agent. For our implementation, we used Python, since this implementation is meant to be a first draft and/or a proof of concept of an implementation of a performance agent. Developing a small program in Python is a lot faster than for example in Java. Further, we based our development on Linux systems, since Linux is used at the GWDG. Our main source for system data is the */proc* directory.

4.1 The */proc* Directory

The */proc* directory is a virtual file system provided by the Linux kernel. The kernel generates the content of the files when they are read. The files themselves do not occupy any space on the disk[14].

4.1.1 */proc/stat*

We will now take a look at */proc/stat*, */proc/meminfo* and */proc/<PID>/stat*. The */proc/stat* directory lists some general information about the system. For our use case,

[14]This can be verified with a *ls -sh* command.

```
cpu  456432 1920 125572 2100094 10029 0 3767 0 0 0
cpu0 115032 498 30002 895477 4938 0 1434 0 0 0
cpu1 111105 542 35050 398895 1589 0 1628 0 0 0
cpu2 115184 480 29943 403243 1714 0 137 0 0 0
cpu3 115111 398 30576 402478 1787 0 568 0 0 0
intr 24289740 ...
ctxt 71526417
btime 1679493930
processes 186059
procs_running 1
procs_blocked 0
softirq 19025529 6933211 1800113  ...
```

Listing 4: Truncated example output of the *cat /proc* command.

it is also important that it keeps track of how much time a CPU core is running in kernel/user mode, idle, or waiting for IO read/writes.

A sample output of *cat /proc/stat* on our Ubuntu system is shown in 4.1.1. We pruned some values of the *intr* and *softirq* rows.

The values of the *cpu[0123]?* row are to be interpreted as follows and describe the utilization of the Central Processing Unit (CPU).

```
cpu user nice system idle iowait irq softirq steal guest guest_nice
```

- *cpu* is simply the name and specifies if the total utilization is shown (*cpu* without suffix). *cpu* is the total of *cpu0 cpu1 .... cpu0*, *cpu1*, ... and show the utilization grouped by the corresponding CPU cores.
- *user* indicates the time the CPU spent in the user mode.
- *nice* presents the time the CPU spent in the user mode with low priority.
- *system* shows the time the CPU spent in the kernel mode.
- *idle* is the time the CPU spend idling.
- *iowait* roughly indicates the time a CPU spent waiting for IO operations to finish.
- *irq* is the time the CPU spent handling interrupts.
- *softirq* is the time the CPU spent handling softirq. In short, routines are called softirq after finishing an interrupt handle when the kernel looks for other occurred interrupts.
- *steal* represents the time spent in other Operating Systems (OSs) (virtualized environments).
- *guest* shows the time spent running a virtual CPU.
- *guest_ nice* is the time spent running a niced guest.

The values are so-called *jiffies* (clock ticks) and represent how many cycles of 1/100 per second are spent in the respective mode [CRK05; TB15].

To calculate the CPU utilization in our programs, we first took the values in an interval of t seconds. Then, we took the difference and multiplied it with $SC\_CLK\_TCK * t$ to get the utilization. The SC_CLK_TCK is often 100 and can be found using Python, as shown in 5.

```
1 def sc_clk_tck():
2     return os.sysconf(os.sysconf_names['SC_CLK_TCK'])
```

Listing 5: Reading the 'SC_CLK_TCK'value from Linux using Python.

intr: shows which interrupt got handled since the last boot. The first value represents the total of serviced interrupts. Depending on the use case, a look into the */proc/interrupts* file can be helpful [CRK05].

ctxt: counts how many context switches are made since the last system boot. A context switch is the process of switching from the currently running process, storing its state and loading the state of the new process [TB15].

btime: shows the boot time since 1970-01-01 00:00:00 + 0000 (UTC) in seconds.

processes: number of processes created since the last boot.

procs_running: number of processes which are currently in a runnable state.

procs_blocked: number of processes blocked because they wait for IO to finish.

softiqr: lists the number of softiqr handled. The first value is the total of all subsequent values, and all subsequent values correspond to their softiqr number.

4.1.2 */proc/<PID>/stat*

So far, we have considered system-wide statistics. Now we want to take a look at process-specific values. For this, we examine the */proc/<PID>/stat* file. Please replace the *<PID>* with an existing Process ID (PID).

The output from this file is less readable and is meant to be used by programs. From the man page[15], we know that there are 52 values, but this would go beyond the scope of this report. For a complete list, please see the man page. We will only take a look at some selected values.

- (1) PID[16] of the selected process.
- (2) The (file-) name of the process.

[15] *man proc*

[16] At this point, it can seem unnecessary to specify the PID twice, once in the path and once in the file. But the path */proc/self* exists, which redirects to */proc/<PIS>*, the directory of the current process.

```
253051 (bash) S 252970 253051 253051 34816 254899 4194304 4435 44153 0 0
4 0 21 18 20 0 1 0 1814020 13402112 1477 18446744073709551615
94339739869184 94339740782349 140728151514896 0 0 0 65536 3686404
1266761467 1 0 0 17 2 0 0 0 0 0 94339741026992 94339741075024
94339760603136 140728151517360 140728151517365 140728151517365
140728151519214 0
```

Listing 6: Example output of the *cat /proc/<PID>/stat* command.

- (3) Current state the process is in. The most important values are (R)unning and (S)leeping.
- (4) *PPID*, the PID of the parent process.
- (14) *utime* is the time the process spent in user mode.
- (15) *stime* is the time the process spent in system/kernel mode.
- (20) *num_threads* is the number of threads belonging to this process.
- (22) *starttime* is the number of clock ticks before the process was started counting from the boot time.

Note that *utime*, *stime*, and *starttime* are recorded using clock ticks. This is discussed in 4.1.1.

4.1.3 */proc/meminfo*

The last file we want to inspect is */proc/meminfo*. As the name suggests, it covers memory information. This file is nicely structured, with key-value-unit-pairs. The values are quite self-describing, and are nice to parse with a simple routine. We do not want to dive deeper into the meaning of each and every value. For more details, please look at the man page. We use this file to gather memory-specific information. Note that the values can be hard to interpret since processes can share their memory space. For example, the total memory available could be lower than the memory used in total by all processes together.

4.2 The implementation

One important design choice was to use the *async* functionality from Python. With that, gathering each metric was able to be implemented with its own subroutine using a loop. The loops then suspended their work using `await asyncio.sleep(sleep)` to pass the execution to the other routines. With this approach, it is easy to extend the functionality of the program without manipulating other functions.

As a drawback, it should be noted that in a more extendable approach, it would be nice to have each metric collection have its own python file/module, which would be loaded dynamically as needed. So far, our implementation has all collection routines in a couple of files linked together. With such an approach, it would be easier to support another tool to collect additional metrics.

Further, our implementation uses the REST-API from the InfluxDB. The developers of InfluxDB provide two Python modules to interact with their DB. Version v1 is archived but still in the PyPi[17] index. Version v2[18] is actively maintained on GitHub.

During the development, we simply could not get them to run as expected, so we fell back to the REST-API, which worked flawlessly using the Influx one-line-protocol. In future development, more work should be put into getting the modules to work since their interface seemed to be well designed.

4.2.1 Usage

Lastly, we want to show some example usages of our tool. Our tool can be downloaded from Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) GitLab instance here[19].

The Tool is run by the command `python main.py`. With the flags `--influx-domain` (default: localhost) `--influx-port` (default: 8086) and `--influx-db` (default: TMP) the respected Influx parameter can be set. Alternatively, the `--local` flag can be used and the metrics are printed to stdout. The tool uses the Influx one-line protocol.

With the flags `--cpu` and `--mem`, general system information about the cpu and the memory usage are collected. Those two flags use the file */proc/stat* and */proc/meminfo*, respectively. When the program is called with a PID, it collects the specific information from */proc/<pid>/stat* file. To specify the collection interval, `--seconds` or `-s` can be used. Their arguments are integers specifying the interval length in seconds. So far, the interval can only be specified using seconds.

Some drawbacks of the tool so far are that a lot of the possible errors are not handled and just passed to the user. The tool does not have an installation or a package to easily install it using a packet manager.

```
python3.10 main.py --cpu --local
python3.10 main.py --cpu --mem --local -s 2
python3.10 main.py --local 390121
python3.10 main.py --influx-domain localhost \
                   --influx-port 8080 \
                   --influx-db TMP \
                   --cpu --mem
```

Listing 7: Usage examples of our implementation.

5 Conclusion

First of all, we discussed the tools cc-metric-collector, PCP and Telegraf. We tried to understand the tools as well as we could. In addition, we explained the Influx one-line protocol which we also support in our implementation. And finally, we got to the implementation of our approach of a performance agent. We discussed the main design choices and showed some examples of how to use it.

[17] `https://pypi.org/`

[18] `https://github.com/influxdata/influxdb-client-python`

[19] `https://gitlab.gwdg.de/lukas.steinegger/performance-agent`

6 Limitations and Future work

Now we want to discuss some limitations of our work as well as some possible future work. We mainly tested the tools only on one Ubuntu machine. Especially our own implementation was developed and tested on the same machine. Running the software on other machines could result in unexpected errors. Additionally, a monitoring system should be set up and tested on multiple machines.

Furthermore, we did not measure the performance of the tools or even compare them to each other. Particularly, it would be interesting to see how our Python implementation performs compared to the other tools. Lastly, it is important to carefully consider the right metrics to collect. Just collecting any metrics can result in a huge amount of data, and the selection process of meaningful metrics is just deferred to the analysis.

References

[CRK05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. 3rd ed. O'Reilly Media, Inc., 2005. ISBN: 9780596005900.

[TB15] A.S. Tanenbaum and H.J. Bos. *Modern Operating Systems, 4th Edition*. English. Pearson Higher Education, 2015. ISBN: 978-0133591620.